HITACHI
Inspire the Next

OpenTP1 Version 7

Programming Guide

3000-3-D51-30(E)

■ Relevant program products

Note: In the program products listed below, those marked with an asterisk (*) might be released later than the other program products.

For AIX 5L V5.1, AIX 5L V5.2, AIX 5L V5.3, and AIX V6.1

P-1M64-2131 uCosminexus TP1/Server Base 07-03*

P-1M64-2331 uCosminexus TP1/FS/Direct Access 07-03*

P-1M64-2431 uCosminexus TP1/FS/Table Access 07-03*

P-1M64-2531 uCosminexus TP1/Client/W 07-02

P-1M64-2631 uCosminexus TP1/Offline Tester 07-00

P-1M64-2731 uCosminexus TP1/Online Tester 07-00

P-1M64-2831 uCosminexus TP1/Multi 07-00

P-1M64-2931 uCosminexus TP1/High Availability 07-00

P-1M64-3131 uCosminexus TP1/Message Control 07-03

P-1M64-3231 uCosminexus TP1/NET/Library 07-04

P-1M64-8131 uCosminexus TP1/Shared Table Access 07-00

P-1M64-8331 uCosminexus TP1/Resource Manager Monitor 07-00

P-1M64-8531 uCosminexus TP1/Extension 1 07-00

P-1M64-C371 uCosminexus TP1/Message Queue 07-01

P-1M64-C771 uCosminexus TP1/Message Queue - Access 07-01

P-F1M64-31311 uCosminexus TP1/Message Control/Tester 07-00

P-F1M64-32311 uCosminexus TP1/NET/User Agent 07-00

P-F1M64-32312 uCosminexus TP1/NET/HDLC 07-00

P-F1M64-32313 uCosminexus TP1/NET/X25 07-00

P-F1M64-32314 uCosminexus TP1/NET/OSI-TP 07-00

P-F1M64-32315 uCosminexus TP1/NET/XMAP3 07-01

P-F1M64-32316 uCosminexus TP1/NET/HSC 07-00

P-F1M64-32317 uCosminexus TP1/NET/NCSB 07-00

P-F1M64-32318 uCosminexus TP1/NET/OSAS-NIF 07-01

P-F1M64-3231B uCosminexus TP1/NET/Secondary Logical Unit - TypeP2 07-00

P-F1M64-3231C uCosminexus TP1/NET/TCP/IP 07-02

P-F1M64-3231D uCosminexus TP1/NET/High Availability 07-00

P-F1M64-3231U uCosminexus TP1/NET/User Datagram Protocol 07-00

R-1M45F-31 uCosminexus TP1/Web 07-00

For AIX 5L V5.3 and AIX V6.1

P-1M64-1111 uCosminexus TP1/Server Base(64) 07-03*

P-1M64-1311 uCosminexus TP1/FS/Direct Access(64) 07-03*

P-1M64-1411 uCosminexus TP1/FS/Table Access(64) 07-03*

P-1M64-1911 uCosminexus TP1/High Availability(64) 07-00

P-1M64-1L11 uCosminexus TP1/Extension 1(64) 07-00

For HP-UX 11i V1 (PA-RISC) and HP-UX 11i V2 (PA-RISC)

P-1B64-3F31 uCosminexus TP1/NET/High Availability 07-00

P-1B64-8531 uCosminexus TP1/Extension 1 07-00

P-1B64-8931 uCosminexus TP1/High Availability 07-00

R-18451-41K uCosminexus TP1/Client/W 07-00

R-18452-41K uCosminexus TP1/Server Base 07-00

R-18453-41K uCosminexus TP1/FS/Direct Access 07-00
R-18454-41K uCosminexus TP1/FS/Table Access 07-00
R-18455-41K uCosminexus TP1/Message Control 07-03*
R-18456-41K uCosminexus TP1/NET/Library 07-04*
R-18459-41K uCosminexus TP1/Offline Tester 07-00
R-1845A-41K uCosminexus TP1/Online Tester 07-00
R-1845C-41K uCosminexus TP1/Shared Table Access 07-00
R-1845D-41K uCosminexus TP1/Resource Manager Monitor 07-00
R-1845E-41K uCosminexus TP1/Multi 07-00
R-1845F-41K uCosminexus TP1/Web 07-00
R-F18455-411K uCosminexus TP1/Message Control/Tester 07-00
R-F18456-411K uCosminexus TP1/NET/User Agent 07-00
R-F18456-415K uCosminexus TP1/NET/XMAP3 07-01*
R-F18456-41CK uCosminexus TP1/NET/TCP/IP 07-02*
For HP-UX 11i V2 (IPF) and HP-UX 11i V3 (IPF)
P-1J64-3F21 uCosminexus TP1/NET/High Availability 07-00
P-1J64-4F11 uCosminexus TP1/NET/High Availability(64) 07-00
P-1J64-8521 uCosminexus TP1/Extension 1 07-00
P-1J64-8611 uCosminexus TP1/Extension 1(64) 07-00
P-1J64-8921 uCosminexus TP1/High Availability 07-00
P-1J64-8A11 uCosminexus TP1/High Availability(64) 07-00
P-1J64-C371 uCosminexus TP1/Message Queue 07-01
P-1J64-C571 uCosminexus TP1/Message Queue(64) 07-01
P-1J64-C871 uCosminexus TP1/Message Queue - Access(64) 07-00
R-18451-21J uCosminexus TP1/Client/W 07-02
R-18452-21J uCosminexus TP1/Server Base 07-03*
R-18453-21J uCosminexus TP1/FS/Direct Access 07-03*
R-18454-21J uCosminexus TP1/FS/Table Access 07-03*
R-18455-21J uCosminexus TP1/Message Control 07-03*
R-18456-21J uCosminexus TP1/NET/Library 07-04*
R-18459-21J uCosminexus TP1/Offline Tester 07-00
R-1845A-21J uCosminexus TP1/Online Tester 07-00
R-1845C-21J uCosminexus TP1/Shared Table Access 07-00
R-1845D-21J uCosminexus TP1/Resource Manager Monitor 07-00
R-1845E-21J uCosminexus TP1/Multi 07-00
R-1845F-21J uCosminexus TP1/Web 07-00
R-1B451-11J uCosminexus TP1/Client/W(64) 07-02
R-1B452-11J uCosminexus TP1/Server Base(64) 07-03*
R-1B453-11J uCosminexus TP1/FS/Direct Access(64) 07-03*
R-1B454-11J uCosminexus TP1/FS/Table Access(64) 07-03*
R-1B455-11J uCosminexus TP1/Message Control(64) 07-03*
R-1B456-11J uCosminexus TP1/NET/Library(64) 07-04*
R-F18455-211J uCosminexus TP1/Message Control/Tester 07-00
R-F18456-215J uCosminexus TP1/NET/XMAP3 07-01*

R-F18456-21CJ uCosminexus TP1/NET/TCP/IP 07-02*

R-F1B456-11CJ uCosminexus TP1/NET/TCP/IP(64) 07-02*

For Solaris 8, Solaris 9, and Solaris 10

P-9D64-3F31 uCosminexus TP1/NET/High Availability 07-00

P-9D64-8531 uCosminexus TP1/Extension 1 07-00

P-9D64-8931 uCosminexus TP1/High Availability 07-00

R-19451-216 uCosminexus TP1/Client/W 07-00

R-19452-216 uCosminexus TP1/Server Base 07-00

R-19453-216 uCosminexus TP1/FS/Direct Access 07-00

R-19454-216 uCosminexus TP1/FS/Table Access 07-00

R-19455-216 uCosminexus TP1/Message Control 07-03*

R-19456-216 uCosminexus TP1/NET/Library 07-04*

R-19459-216 uCosminexus TP1/Offline Tester 07-00

R-1945A-216 uCosminexus TP1/Online Tester 07-00

R-1945C-216 uCosminexus TP1/Shared Table Access 07-00

R-1945D-216 uCosminexus TP1/Resource Manager Monitor 07-00

R-1945E-216 uCosminexus TP1/Multi 07-00

R-F19456-2156 uCosminexus TP1/NET/XMAP3 07-01*

R-F19456-21C6 uCosminexus TP1/NET/TCP/IP 07-02*

For Red Hat Enterprise Linux AS 4 (AMD64 & Intel EM64T), Red Hat Enterprise Linux AS 4 (x86), Red Hat Enterprise Linux ES 4 (AMD64 & Intel EM64T), and Red Hat Enterprise Linux ES 4 (x86)

P-9S64-2161 uCosminexus TP1/Server Base 07-00

P-9S64-2351 uCosminexus TP1/FS/Direct Access 07-00

P-9S64-2451 uCosminexus TP1/FS/Table Access 07-00

P-9S64-2551 uCosminexus TP1/Client/W 07-00

P-9S64-3151 uCosminexus TP1/Message Control 07-00

P-9S64-3251 uCosminexus TP1/NET/Library 07-00

P-9S64-C371 uCosminexus TP1/Message Queue 07-01

P-F9S64-3251C uCosminexus TP1/NET/TCP/IP 07-00

P-F9S64-3251U uCosminexus TP1/NET/User Datagram Protocol 07-00

R-1845F-A15 uCosminexus TP1/Web 07-00

For Red Hat Enterprise Linux AS 4 (AMD64 & Intel EM64T), Red Hat Enterprise Linux AS 4 (x86), Red Hat Enterprise Linux ES 4 (AMD64 & Intel EM64T), Red Hat Enterprise Linux ES 4 (x86), Red Hat Enterprise Linux 5 (AMD/Intel 64), Red Hat Enterprise Linux 5 (x86), Red Hat Enterprise Linux 5 Advanced Platform (AMD/Intel 64), and Red Hat Enterprise Linux 5 Advanced Platform (x86)

P-9S64-2951 uCosminexus TP1/High Availability 07-00

P-9S64-8551 uCosminexus TP1/Extension 1 07-00

P-9S64-C771 uCosminexus TP1/Message Queue - Access 07-01

P-F9S64-3251D uCosminexus TP1/NET/High Availability 07-00

For Red Hat Enterprise Linux 5 (AMD/Intel 64), Red Hat Enterprise Linux 5 (x86), Red Hat Enterprise Linux 5 Advanced Platform (AMD/Intel 64), and Red Hat Enterprise Linux 5 Advanced Platform (x86)

P-9S64-2171 uCosminexus TP1/Server Base 07-03

P-9S64-2361 uCosminexus TP1/FS/Direct Access 07-03

P-9S64-2461 uCosminexus TP1/FS/Table Access 07-03

P-9S64-2561 uCosminexus TP1/Client/W 07-02

P-9S64-3161 uCosminexus TP1/Message Control 07-03*

P-9S64-3261 uCosminexus TP1/NET/Library 07-04*

P-9S64-C571 uCosminexus TP1/Message Queue 07-01

P-F9S64-32611 uCosminexus TP1/NET/User Agent 07-00

P-F9S64-3261C uCosminexus TP1/NET/TCP/IP 07-02

P-F9S64-3261U uCosminexus TP1/NET/User Datagram Protocol 07-00

For Red Hat Enterprise Linux 5 (AMD/Intel 64) and Red Hat Enterprise Linux 5 Advanced Platform (AMD/Intel 64)

P-9W64-2111 uCosminexus TP1/Server Base(64) 07-03

P-9W64-2311 uCosminexus TP1/FS/Direct Access(64) 07-03

P-9W64-2411 uCosminexus TP1/FS/Table Access(64) 07-03

P-9W64-2911 uCosminexus TP1/High Availability(64) 07-02

P-9W64-8511 uCosminexus TP1/Extension 1(64) 07-02

For Red Hat Enterprise Linux AS 4 (IPF)

P-9V64-2121 uCosminexus TP1/Server Base 07-00

P-9V64-2321 uCosminexus TP1/FS/Direct Access 07-00

P-9V64-2421 uCosminexus TP1/FS/Table Access 07-00

P-9V64-2521 uCosminexus TP1/Client/W 07-00

P-9V64-3121 uCosminexus TP1/Message Control 07-00

P-9V64-3221 uCosminexus TP1/NET/Library 07-00

P-9V64-C371 uCosminexus TP1/Message Queue(64) 07-01

P-9V64-C771 uCosminexus TP1/Message Queue - Access(64) 07-00

P-F9V64-3221C uCosminexus TP1/NET/TCP/IP 07-00

P-F9V64-3221U uCosminexus TP1/NET/User Datagram Protocol 07-00

For Red Hat Enterprise Linux AS 4 (IPF), Red Hat Enterprise Linux 5 (Intel Itanium), and Red Hat Enterprise Linux 5 Advanced Platform (Intel Itanium)

P-9V64-2921 uCosminexus TP1/High Availability 07-00

P-9V64-8521 uCosminexus TP1/Extension 1 07-00

P-F9V64-3221D uCosminexus TP1/NET/High Availability 07-00

For Red Hat Enterprise Linux 5 (Intel Itanium) and Red Hat Enterprise Linux 5 Advanced Platform (Intel Itanium)

P-9V64-2131 uCosminexus TP1/Server Base 07-02

P-9V64-2331 uCosminexus TP1/FS/Direct Access 07-02

P-9V64-2431 uCosminexus TP1/FS/Table Access 07-02

P-9V64-2531 uCosminexus TP1/Client/W 07-02

P-9V64-3131 uCosminexus TP1/Message Control 07-03*

P-9V64-3231 uCosminexus TP1/NET/Library 07-04*

P-F9V64-3231C uCosminexus TP1/NET/TCP/IP 07-02*

P-F9V64-3231U uCosminexus TP1/NET/User Datagram Protocol 07-00

For Windows 2000, Windows Server 2003, Windows Server 2003 x64 Editions, Windows Server 2003 R2, Windows Server 2003 R2 x64 Editions, Windows XP, Windows Vista, and Windows Vista x64

P-2464-2144 uCosminexus TP1/Client/P 07-02

For Windows 2000, Windows Server 2003, Windows Server 2003 x64 Editions, Windows Server 2003 R2, Windows Server 2003 R2 x64 Editions, and Windows XP

R-1845F-8134 uCosminexus TP1/Web 07-00

For Windows 2000, Windows Server 2003, Windows Server 2003 x64 Editions, Windows Server 2003 R2, Windows Server 2003 R2 x64 Editions, Windows XP, Windows Vista, Windows Vista x64, Windows Server 2008, and Windows Server 2008 x64

P-2464-7824 uCosminexus TP1/Client for .NET Framework 07-03

R-15451-21 uCosminexus TP1/Connector for .NET Framework 07-03

For Windows Server 2003, Windows Server 2003 x64 Editions, Windows Server 2003 R2, Windows Server 2003 R2 x64 Editions, Windows XP, Windows Vista, Windows Vista x64, Windows Server 2008, and Windows Server 2008 x64

P-2464-2274 uCosminexus TP1/Server Base 07-03*

P-2464-2374 uCosminexus TP1/FS/Direct Access 07-03*

P-2464-2474 uCosminexus TP1/FS/Table Access 07-03*

P-2464-2544 uCosminexus TP1/Extension 1 07-00

P-2464-3154 uCosminexus TP1/Message Control 07-03*

P-2464-3254 uCosminexus TP1/NET/Library 07-04*

P-2464-3354 uCosminexus TP1/Messaging 07-00

P-2464-C374 uCosminexus TP1/Message Queue 07-01

P-2464-C774 uCosminexus TP1/Message Queue - Access 07-00

P-F2464-3254C uCosminexus TP1/NET/TCP/IP 07-02*

R-15452-21 uCosminexus TP1/Extension for .NET Framework 07-00

R-1945B-24 uCosminexus TP1/LiNK 07-02

For Windows Server 2003, Windows Server 2003 x64 Editions, Windows Server 2003 R2, Windows Server 2003 R2 x64 Editions, and Windows XP

P-F2464-32545 uCosminexus TP1/NET/XMAP3 07-01*

For Windows Server 2003, Windows Server 2003 x64 Editions, Windows Server 2003 R2, Windows Server 2003 R2 x64 Editions, Windows Server 2008, and Windows Server 2008 x64

P-2464-2934 uCosminexus TP1/High Availability 07-00

P-F2464-3254D uCosminexus TP1/NET/High Availability 07-00

For Java VM

P-2464-7394 uCosminexus TP1/Client/J 07-02

P-2464-73A4 uCosminexus TP1/Client/J 07-02

This manual can be used for products other than the products shown above. For details, see the *Release Notes*.

This product was developed under a quality management system that has received ISO9001 and TickIT certification.

■ Trademarks

■ Restrictions

Printed in Japan.

■ Edition history

Edition 1 (3000-3-D51(E)): June 2006

Edition 3 (3000-3-D51-30(E)): October 2010

■ Copyright

Summary of amendments

The following table lists changes in this manual (3000-3-D51-30(E)) and product changes related to this manual for uCosminexus TP1/Server Base 07-03, uCosminexus TP1/Server Base(64) 07-03, uCosminexus TP1/Message Control 07-03, uCosminexus TP1/Message Control(64) 07-03, uCosminexus TP1/NET/Library 07-04, and uCosminexus TP1/NET/Library(64) 07-04

Changes	Location
A note has been added about coding of OpenTP1 UAPs.	*1.3.1(3)*
A note about the `dc_clt_chained_accept_notification` function has been added to the explanation about the CUP on the receiving end.	*2.1.10*
An explanation about global domains has been added.	*2.1.17*
The `prctee` process, used to redirect the standard output and standard error output of OpenTP1, can now be stopped and restarted. With this change, the following command has been added: • `prctctrl`	*2.4.1(1)*
Application timer start request statuses can now be displayed. With this change, the following command has been added: • `mcfalstap`	*2.4.1(1)*
User timer monitoring statuses can now be displayed. With this change, the following command has been added: • `mcftlsutm`	*2.4.1(1)*, *3.8.4*
Explanations have been added about the causes of MCF events that report UAP abnormal terminations and that report discarding of unprocessed messages.	*3.7.1(3)(a)*, *Table 3-16* in *3.10*
An explanation has been added regarding the fact that timer start request messages are immediately discarded and an `ERREVTA` error event is reported when an OpenTP1 normal termination command is executed.	*3.10.5(1)*
The description has been added about an item in the `examples/tools/` directory.	*8.1.2(1)(b)*

The following table lists changes in this manual (3000-3-D51-30(E)) and product changes related to this manual for uCosminexus TP1/Message Control 07-02 and uCosminexus TP1/NET/Library 07-03

Changes	Location
The facility for dynamic loading of service functions can now be used by MHPs.	*1.2.3(1)*, *1.3.4(3)*, *2.1.19*
Library functions can now be used to display the statuses of the MCF communication service and the application start service. With this change, the following functions have been added: • `dc_mcf_tlscom` • `CBLDCMCF('TLSCOM  ')`	*Table 1-2* in *1.4.2(1)*, *Table 1-7* in *1.4.2(2)*, *3.1*

Changes	Location
Library functions can now be used to display connection statuses, and to establish and release connections. With this change, the following functions have been added: • dc_mcf_tactcn • dc_mcf_tdctcn • dc_mcf_tlscn • CBLDCMCF('TACTCN ') • CBLDCMCF('TDCTCN ') • CBLDCMCF('TLSCN ')	*Table 1-2* in *1.4.2(1)*, *Table 1-7* in *1.4.2(2)*, *3.2*, *3.2.1*, *3.2.2*
Library functions can now be used to start and stop reception of server-type connection establishment requests. With this change, the following functions have been added: • dc_mcf_tofln • dc_mcf_tonln • CBLDCMCF('TOFLN ') • CBLDCMCF('TONLN ')	*Table 1-2* in *1.4.2(1)*, *Table 1-7* in *1.4.2(2)*, *3.2.3*
Library functions can be now used to display the reception status of connection establishment requests. With this change, the following functions have been added: • dc_mcf_tlsln • CBLDCMCF('TLSLN ')	*Table 1-2* in *1.4.2(1)*, *Table 1-7* in *1.4.2(2)*, *3.2.3*
Library functions can now be used to delete application timer start requests. With this change, the following functions have been added: • dc_mcf_adltap • CBLDCMCF('ADLTAP ')	*Table 1-2* in *1.4.2(1)*, *Table 1-7* in *1.4.2(2)*, *3.3*
Library functions can now be used to display the status of logical terminals, shut down logical terminals, release the shutdown status of logical terminals, and delete the output queues of logical terminals. With this change, the following functions have been added: • dc_mcf_tactle • dc_mcf_tdctle • dc_mcf_tdlqle • dc_mcf_tlsle • CBLDCMCF('TACTLE ') • CBLDCMCF('TDCTLE ') • CBLDCMCF('TDLQLE ') • CBLDCMCF('TLSLE ')	*Table 1-2* in *1.4.2(1)*, *Table 1-7* in *1.4.2(2)*, *3.4*, *Table 3-16* in *3.10*, *3.10.5*
The network status of messages exchanged with a remote system can now be displayed. With this change, the following command has been added: • mcftlsln	*2.4.1(1)*

Changes	Location
Explanations have been added about the functional differences between functions used in operations and the corresponding operation commands.	*3.1(2)*, *3.2.1(3)*, *3.2.3(1)*, *3.3(2)*, *3.4(4)*
Explanations have been added about the relationship between products that support communications protocols and functions used in operations.	*3.5*

The following table lists changes in this manual (3000-3-D51-30(E)) and product changes related to this manual for uCosminexus TP1/Message Control 07-01 and uCosminexus TP1/NET/Library 07-01

Changes	Location
Starting and stopping the reception of server-type connection establishment requests can now be performed manually. With this change, the following commands have been added: • `mcftofln` • `mcftonln`	*2.4.1(1)*
MCF information can now be acquired as real-time statistical information.	*8.1.2(2)(a)*, *8.10*

In addition to the above changes, minor editorial corrections have been made.

The following table lists changes in the manual (3000-3-D51-20(E)) and product changes related to that manual for uCosminexus TP1/Server Base 07-02, uCosminexus TP1/Message Control 07-01, and uCosminexus TP1/NET/Library 07-01.

Changes
A facility for outputting audit logs has been added. With this addition, a method of acquiring audit logs from a UAP has been added.
A facility for dynamic loading of service functions has been added.
The description of the remote API facility has been changed.

The following table lists changes in the manual (3000-3-D51-20(E)) and product changes related to that manual for uCosminexus TP1/Server Base 07-01.

Changes
A function has been added for displaying the product name, version number, and other information about products operating in environments set up in the OpenTP1directory. With this addition, the `dcpplist` command has been added.

Preface

This manual explains how to create application programs which can be used with the following program products of OpenTP1:

- Distributed transaction processing facility TP1/Server Base
- Distributed application server TP1/LiNK

In this manual, an application program which is created by the user is abbreviated to a User Application Program (UAP).

Products described in this manual, other than those for which the manual is released, may not work with OpenTP1 Version 7 products. You need to confirm that the products you want to use work with OpenTP1 Version 7 products.

Intended readers

This manual is intended for programmers who create application programs used with TP1/Server Base or TP1/LiNK.

Readers of this manual are assumed to have knowledge about operating systems, online systems, handling of the machine to be used, and the syntax of the high-level language (C, C++, or COBOL) used for coding application programs.

This manual assumes that the reader has read the *OpenTP1 Description* manual or *TP1/LiNK User's Guide* manual.

Organization of this manual

This manual is organized into the following chapters and appendixes:

1. OpenTP1 Application Programs

This chapter outlines application programs used with OpenTP1.

2. Basic OpenTP1 Facilities (TP1/Server Base, TP1/LiNK)

This chapter explains the facilities available with application programs which run at nodes comprising only the base product of OpenTP1 system, TP1/Server Base or TP1/LiNK.

3. Facilities Provided by TP1/Message Control

This chapter explains the facilities available with application programs which run at nodes where the product for message exchanging mode communication, TP1/Message Control, is installed in the system.

4. Facilities for User Data

This chapter explains how to use various user files with OpenTP1 application programs.

5. X/Open-compliant Application Programming Interface

This chapter explains the X/Open specification which is useful for OpenTP1 application programs.

6. X/Open-compliant Inter-application Communication (TxRPC)

This chapter explains the X/Open specification which can be created as an OpenTP1 application program (TxRPC interface).

7. Facilities Provided by TP1/Multi

This chapter explains the facilities available with application programs which run at nodes where the product TP1/Multi for a cluster/parallel mode OpenTP1 system is installed in the system.

8. OpenTP1 Samples

This chapter explains how to use samples given by the OpenTP1 for easy setup of the system.

A. Output Format of Undecided Transaction Information

This appendix explains the output format of undecided transaction information to be used for analyzing the conditions of transactions which have not been processed successfully.

B. Output Format of Deadlock Information

This appendix explains the output format of deadlock and timeout information which will be output by OpenTP1 when a deadlock between application programs occurs.

C. Examples of System Configurations Requiring Consideration of the Multi-Scheduler Facility

This appendix explains the examples of system configurations for which you should consider the multi-scheduler facility and examples of resolutions. As systems become larger and machines and networks boast increasingly better performances, conventional scheduler may experience difficulty scheduling messages efficiently.

Related publications

This manual is part of a related set of manuals. The manuals in the set are listed below (with the manual numbers):

OpenTP1 products

- *OpenTP1 Version 7 Description* (3000-3-D50(E))
- *OpenTP1 Version 7 Programming Guide* (3000-3-D51(E))
- *OpenTP1 Version 7 System Definition* (3000-3-D52(E))
- *OpenTP1 Version 7 Operation* (3000-3-D53(E))
- *OpenTP1 Version 7 Programming Reference C Language* (3000-3-D54(E))
- *OpenTP1 Version 7 Programming Reference COBOL Language* (3000-3-D55(E))
- *OpenTP1 Version 7 Messages* (3000-3-D56(E))
- *OpenTP1 Version 7 Tester and UAP Trace User's Guide* (3000-3-D57(E))
- *OpenTP1 Version 7 TP1/Client User's Guide TP1/Client/W, TP1/Client/P* (3000-3-D58(E))
- *OpenTP1 Version 7 TP1/Client User's Guide TP1/Client/J* (3000-3-D59(E))
- *OpenTP1 Version 7 TP1/LiNK User's Guide* (3000-3-D60(E))#
- *OpenTP1 Version 7 Protocol TP1/NET/TCP/IP* (3000-3-D70(E))
- *OpenTP1 Version 7 TP1/Message Queue User's Guide* (3000-3-D90(E))#
- *OpenTP1 Version 7 TP1/Message Queue Messages* (3000-3-D91(E))#
- *OpenTP1 Version 7 TP1/Message Queue Application Programming Guide* (3000-3-D92(E))#
- *OpenTP1 Version 7 TP1/Message Queue Application Programming Reference* (3000-3-D93(E))#

Other OpenTP1 products

- *TP1/Web User's Guide and Reference* (3000-3-D62(E))#

Other related products

- *Indexed Sequential Access Method ISAM* (3000-3-046(E))
- *XP/W* (3000-3-047(E))
- *Extended Mapping Service 2/Workstation XMAP2/W DESCRIPTION/USER'S GUIDE* (3000-7-421(E))
- *SEWB 3 General Information* (3000-7-450(E))
- *Job Management Partner 1/Base User's Guide* (3020-3-K06(E))

- *Job Management Partner 1/Base Messages* (3020-3-K07(E))
- *Job Management Partner 1/Base Software Developer's Guide* (3020-3-K08(E))

For OpenTP1 protocol manuals, please check whether English versions are available.

#

If you want to use this manual, confirm that it has been published. (Some of these manuals might not have been published yet.)

Reference manuals for using TP1/Message Control (message exchanging facility)

The *OpenTP1 Version 6 Programming Reference* manual does not contain information about syntax which, when TP1/Message Control is in use, is specific to products supporting the communication protocol. For the syntax of the following APIs, see the *OpenTP1 Protocol* manual in the version for the pertinent protocol:

- Receive a message (`dc_mcf_receive()`, `CBLDCMCF('RECEIVE ')`)
- Receive a synchronous message (`dc_mcf_recvsync()`, `CBLDCMCF('RECVSYNC')`)
- Send a response message (`dc_mcf_reply()`, `CBLDCMCF('REPLY   ')`)
- Resend a message (`dc_mcf_resend()`, `CBLDCMCF('RESEND  ')`)
- Send a message (`dc_mcf_send()`, `CBLDCMCF('SEND    ')`)
- Exchange a synchronous message (`dc_mcf_sendrecv()`, `CBLDCMCF('SENDRECV')`)
- Send a synchronous message (`dc_mcf_sendsync()`, `CBLDCMCF('SENDSYNC')`)

Conventions: Abbreviations for product names

This manual uses the following abbreviations for product names:

Abbreviation		Full name or meaning
AIX		AIX 5L V5.1
		AIX 5L V5.2
		AIX 5L V5.3
		AIX V6.1
Client .NET	TP1/Client for .NET Framework	uCosminexus TP1/Client for .NET Framework
Connector .NET	TP1/Connector for .NET Framework	uCosminexus TP1/Connector for .NET Framework

Abbreviation			Full name or meaning
DPM			JP1/ServerConductor/Deployment Manager
HI-UX/WE2			HI-UX/workstation Extended Version 2
HP-UX	HP-UX (IPF)		HP-UX 11i V2 (IPF)
			HP-UX 11i V3 (IPF)
	HP-UX (PA-RISC)		HP-UX 11i V1 (PA-RISC)
			HP-UX 11i V2 (PA-RISC)
IPF			Itanium(R) Processor Family
Java			Java™
JP1	JP1/AJS2	JP1/AJS2 - Agent	JP1/Automatic Job Management System 2 - Agent
		JP1/AJS2 - Manager	JP1/Automatic Job Management System 2 - Manager
		JP1/AJS2 - View	JP1/Automatic Job Management System 2 - View
	JP1/AJS2 - Scenario Operation	JP1/AJS2 - Scenario Operation Manager	JP1/Automatic Job Management System 2 - Scenario Operation Manager
		JP1/AJS2 - Scenario Operation View	JP1/Automatic Job Management System 2 - Scenario Operation View
		JP1/NETM/Audit	JP1/NETM/Audit - Manager
Linux			Linux(R)
Linux (AMD64/Intel EM64T/x86)			Red Hat Enterprise Linux AS 4 (AMD64 & Intel EM64T)
			Red Hat Enterprise Linux AS 4 (x86)
			Red Hat Enterprise Linux ES 4 (AMD64 & Intel EM64T)
			Red Hat Enterprise Linux ES 4 (x86)
			Red Hat Enterprise Linux 5 (AMD/Intel 64)
			Red Hat Enterprise Linux 5 (x86)
			Red Hat Enterprise Linux 5 Advanced Platform (AMD/Intel 64)
			Red Hat Enterprise Linux 5 Advanced Platform (x86)
Linux (IPF)			Red Hat Enterprise Linux AS 4 (IPF)

Abbreviation		Full name or meaning
		Red Hat Enterprise Linux 5 (Intel Itanium)
		Red Hat Enterprise Linux 5 Advanced Platform (Intel Itanium)
MS-DOS		Microsoft(R) MS-DOS(R)
NETM/DM		JP1/NETM/DM Client
		JP1/NETM/DM Manager
		JP1/NETM/DM SubManager
Oracle		Oracle 10g
		Oracle9i
Solaris		Solaris 8
		Solaris 9
		Solaris 10
TP1/Client	TP1/Client/J	uCosminexus TP1/Client/J
	TP1/Client/P	uCosminexus TP1/Client/P
	TP1/Client/W	uCosminexus TP1/Client/W
		uCosminexus TP1/Client/W(64)
TP1/EE		uCosminexus TP1/Server Base Enterprise Option
		uCosminexus TP1/Server Base Enterprise Option(64)
TP1/Extension 1		uCosminexus TP1/Extension 1
		uCosminexus TP1/Extension 1(64)
TP1/FS/Direct Access		uCosminexus TP1/FS/Direct Access
		uCosminexus TP1/FS/Direct Access(64)
TP1/FS/Table Access		uCosminexus TP1/FS/Table Access
		uCosminexus TP1/FS/Table Access(64)
TP1/High Availability		uCosminexus TP1/High Availability
		uCosminexus TP1/High Availability(64)
TP1/LiNK		uCosminexus TP1/LiNK

Abbreviation		Full name or meaning
TP1/Message Control		uCosminexus TP1/Message Control
		uCosminexus TP1/Message Control(64)
TP1/Message Control/Tester		uCosminexus TP1/Message Control/Tester
TP1/Message Queue		uCosminexus TP1/Message Queue
		uCosminexus TP1/Message Queue(64)
TP1/Message Queue - Access		uCosminexus TP1/Message Queue - Access
		uCosminexus TP1/Message Queue - Access(64)
TP1/Messaging		uCosminexus TP1/Messaging
TP1/Multi		uCosminexus TP1/Multi
TP1/NET/HDLC		uCosminexus TP1/NET/HDLC
TP1/NET/High Availability		uCosminexus TP1/NET/High Availability
		uCosminexus TP1/NET/High Availability(64)
TP1/NET/HSC		uCosminexus TP1/NET/HSC
TP1/NET/Library		uCosminexus TP1/NET/Library
		uCosminexus TP1/NET/Library(64)
TP1/NET/NCSB		uCosminexus TP1/NET/NCSB
TP1/NET/OSAS-NIF		uCosminexus TP1/NET/OSAS-NIF
TP1/NET/OSI-TP		uCosminexus TP1/NET/OSI-TP
TP1/NET/SLU - TypeP2	TP1/NET/ Secondary Logical Unit - TypeP2	uCosminexus TP1/NET/Secondary Logical Unit - TypeP2
TP1/NET/TCP/IP		uCosminexus TP1/NET/TCP/IP
		uCosminexus TP1/NET/TCP/IP(64)
TP1/NET/UDP		uCosminexus TP1/NET/User Datagram Protocol
TP1/NET/User Agent		uCosminexus TP1/NET/User Agent
TP1/NET/X25		uCosminexus TP1/NET/X25
TP1/NET/X25-Extended		uCosminexus TP1/NET/X25-Extended
TP1/NET/XMAP3		uCosminexus TP1/NET/XMAP3

Abbreviation	Full name or meaning
TP1/Offline Tester	uCosminexus TP1/Offline Tester
TP1/Online Tester	uCosminexus TP1/Online Tester
TP1/Resource Manager Monitor	uCosminexus TP1/Resource Manager Monitor
TP1/Server Base	uCosminexus TP1/Server Base
	uCosminexus TP1/Server Base(64)
TP1/Shared Table Access	uCosminexus TP1/Shared Table Access
TP1/Web	uCosminexus TP1/Web
Windows 2000	Microsoft(R) Windows(R) 2000 Advanced Server Operating System
	Microsoft(R) Windows(R) 2000 Datacenter Server Operating System
	Microsoft(R) Windows(R) 2000 Professional Operating System
	Microsoft(R) Windows(R) 2000 Server Operating System
Windows Server 2003	Microsoft(R) Windows Server(R) 2003, Datacenter Edition
	Microsoft(R) Windows Server(R) 2003, Enterprise Edition
	Microsoft(R) Windows Server(R) 2003, Standard Edition
Windows Server 2003 R2	Microsoft(R) Windows Server(R) 2003 R2, Enterprise Edition
	Microsoft(R) Windows Server(R) 2003 R2, Standard Edition
Windows Server 2003 x64 Editions	Microsoft(R) Windows Server(R) 2003, Datacenter x64 Edition
	Microsoft(R) Windows Server(R) 2003, Enterprise x64 Edition
	Microsoft(R) Windows Server(R) 2003, Standard x64 Edition
Windows Server 2003 R2 x64 Editions	Microsoft(R) Windows Server(R) 2003 R2, Enterprise x64 Edition
	Microsoft(R) Windows Server(R) 2003 R2, Standard x64 Edition
Windows Server 2008	Microsoft(R) Windows Server(R) 2008 Datacenter (x86)

Abbreviation	Full name or meaning
	Microsoft(R) Windows Server(R) 2008 Enterprise (x86)
	Microsoft(R) Windows Server(R) 2008 Standard (x86)
Windows Server 2008 x64 Editions	Microsoft(R) Windows Server(R) 2008 Datacenter (x64)
	Microsoft(R) Windows Server(R) 2008 Enterprise (x64)
	Microsoft(R) Windows Server(R) 2008 Standard (x64)
Windows Vista	Microsoft(R) Windows Vista(R) Business (x86)
	Microsoft(R) Windows Vista(R) Enterprise (x86)
	Microsoft(R) Windows Vista(R) Ultimate (x86)
Windows Vista x64 Editions	Microsoft(R) Windows Vista(R) Business (x64)
	Microsoft(R) Windows Vista(R) Enterprise (x64)
	Microsoft(R) Windows Vista(R) Ultimate (x64)
Windows XP	Microsoft(R) Windows(R) XP Professional Operating System

- If there is no difference in OS functionality, the term *Windows* is used to indicate Windows 2000, Windows Server 2003, Windows Server 2008, Windows XP, and Windows Vista.
- The term *UNIX* is used to indicate AIX, HP-UX, Linux, and Solaris.

Conventions: Acronyms

This manual also uses the following acronyms:

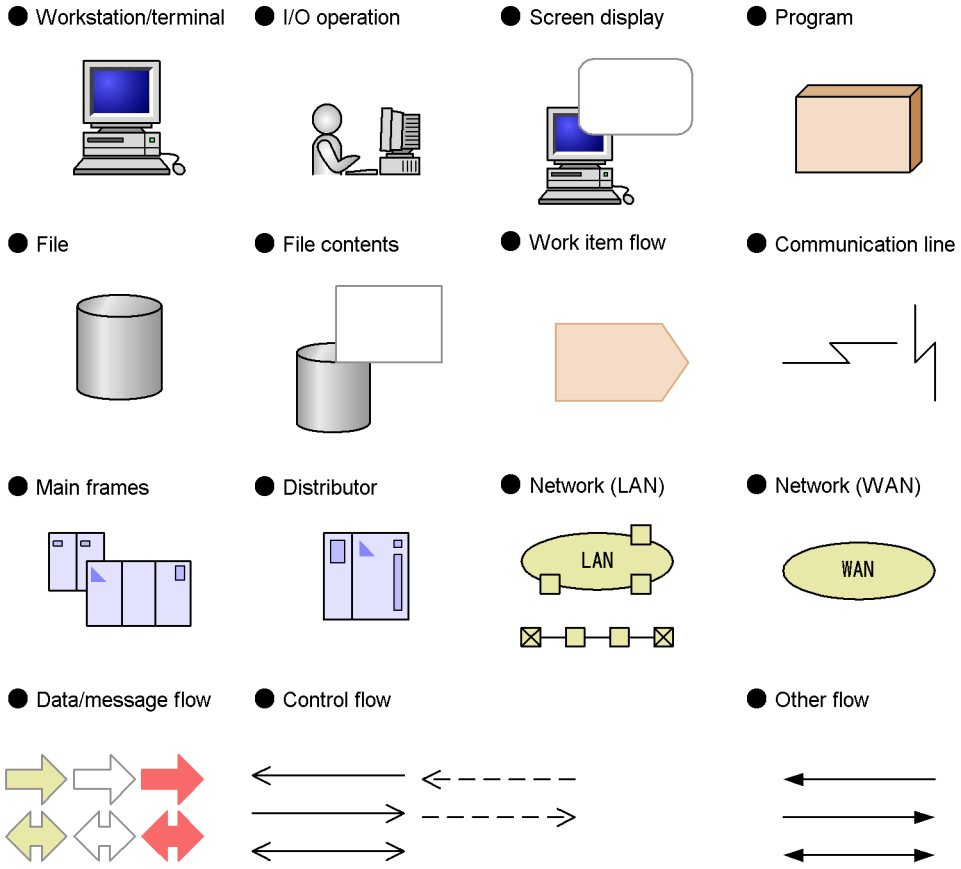
Acronym	Full name or meaning
ANSI	American National Standards Institute
AP	Application Program
API	Application Programming Interface
CPU	Central Processing Unit
CRM	Communication Resource Manager
CUP	Client User Program
DAM	Direct Access Method

Acronym	Full name or meaning
DBMS	Database Management System
DCE	Distributed Computing Environment
DML	Data Manipulation Language
DNS	Domain Name System
GUI	Graphical User Interface
HA	High Availability
I/O	Input/Output
ID	Identifier
IDL	Interface Definition Language
ISAM	Indexed Sequential Access Method
IST	Internode Shared Table
LAN	Local Area Network
MCF	Message Control Facility
MHP	Message Handling Program
MQI	Message Queue Interface
OS	Operating System
OSI	Open Systems Interconnection
OSI TP	Open Systems Interconnection Transaction Processing
PC	Personal Computer
PRF	Performance
RM	Resource Manager
RPC	Remote Procedure Call
RTS	Real Time Statistic
SPP	Service Providing Program
SUP	Service Using Program
TAM	Table Access Method
TCP/IP	Transmission Control Protocol/Internet Protocol

Acronym	Full name or meaning
UAP	User Application Program
UOC	User Own Coding
VM	Virtual Machine
WAN	Wide Area Network
WS	Workstation
XA	Extended Architecture
XAR	Extended Architecture Resource

Conventions: Diagrams

This manual uses the following conventions in diagrams:



Conventions: Differences between JIS and ASCII keyboards

The JIS code and ASCII code keyboards are different in the input characters represented by the following codes. In this manual, the use of a JIS keyboard is assumed for these characters.

Code	JIS keyboard	ASCII keyboard
$(5c)_{16}$	¥ (yen symbol)	\ (backslash)
$(7e)_{16}$	‾ (overline)	~ (tilde)

Conventions: Differences in installation directory paths

This manual uses the notation /BeTRAN to indicate the OpenTP1 installation directory.

The actual installation directory differs depending on the operating system. Use the following table to determine the actual installation directory for your OS.

As written in this manual	Actual directory for each OS		
	AIX, HP-UX, and Solaris	Linux	Windows
/BeTRAN	/BeTRAN	/opt/OpenTP1	The directory in which OpenTP1 was installed

Conventions: Fonts and symbols

The following table explains the fonts used in this manual:

Font	Convention
Bold	**Bold** type indicates text on a window, other than the window title. Such text includes menus, menu options, buttons, radio box options, or explanatory labels. For example: • From the **File** menu, choose **Open**. • Click the **Cancel** button. • In the **Enter name** entry box, type your name.
Italics	*Italics* are used to indicate a placeholder for some actual text to be provided by the user or system. For example: • Write the command as follows: `copy` *source-file target-file* • The following message appears: `A file was not found. (file =` *file-name*`)` *Italics* are also used for emphasis. For example: • Do *not* delete the configuration file.
`Code font`	A `code font` indicates text that the user enters without change, or text (such as messages) output by the system. For example: • At the prompt, enter `dir`. • Use the `send` command to send mail. • The following message is displayed: `The password is incorrect.`

The following table explains the symbols used in this manual:

Symbol	Convention
|	In syntax explanations, a vertical bar separates multiple items, and has the meaning of OR. For example: `A|B|C` means A, or B, or C.
{ }	In syntax explanations, curly brackets indicate that only one of the enclosed items is to be selected. For example: `{A|B|C}` means only one of A, or B, or C.

Symbol	Convention
[ ]	In syntax explanations, square brackets indicate that the enclosed item or items are optional. For example: [A] means that you can specify A or nothing. [B|C] means that you can specify B, or C, or nothing.
. . .	In coding, an ellipsis (...) indicates that one or more lines of coding are not shown for purposes of brevity. In syntax explanations, an ellipsis indicates that the immediately preceding item can be repeated as many times as necessary. For example: A, B, B, . . . means that, after you specify A, B, you can specify B as many times as necessary.

Conventions: KB, MB, GB, and TB

This manual uses the following conventions:

- 1 KB (kilobyte) is 1,024 bytes.
- 1 MB (megabyte) is $1{,}024^2$ bytes.
- 1 GB (gigabyte) is $1{,}024^3$ bytes.
- 1 TB (terabyte) is $1{,}024^4$ bytes.

Conventions: Platform-specific notational differences

For the Windows version of OpenTP1, there are some notational differences from the description in the manual. The following table describes these differences.

Item	Description in the manual	Change to:
Environment variable	$*aaaaaa* Example: $DCDIR	%*aaaaaa*% Example: %DCDIR%
Path name separator	Colon (:)	Semicolon (;)
Directory name separator	Slash (/)	Backslash (\)
Absolute path name	A path from the root directory Example: /tmp	A path name from a drive letter and the root directory Example: C:\tmp
Executable file name	File name only (without an extension) Example: mcfmngrd	File name with an extension Example: mcfmngrd.exe
make command	make	nmake

Conventions: Version numbers

The version numbers of Hitachi program products are usually written as two sets of two digits each, separated by a hyphen. For example:

- Version 1.00 (or 1.0) is written as 01-00.
- Version 2.05 is written as 02-05.
- Version 2.50 (or 2.5) is written as 02-50.
- Version 12.25 is written as 12-25.

The version number might be shown on the spine of a manual as *Ver. 2.00*, but the same version number would be written in the program as *02-00*.

Acknowledgments

Quotations from X/Open CAE Specification Distributed Transaction Processing: The XATMI Specification published by X/Open Company Limited

The specification as interpreted in the above document is quoted in the following section of this manual to give information about the usage with OpenTP1:

Chapter *5. X/Open-Compliant Application Programming Interface*

5.1 XATMI Interface (Client/Server Mode Communication)

Quotations from X/Open CAE Specification Distributed Transaction Processing: The TX (Transaction Demarcation) Specification published by X/Open Company Limited

The specification as interpreted in the above document is quoted in the following section of this manual to give information about the usage with OpenTP1:

Chapter *5. X/Open-Compliant Application Programming Interface*

5.2 TX Interface (Transaction Control)

Quotation from X/Open Preliminary Specification Distributed Transaction Processing: The TxRPC Specification published by X/Open Company Limited

The specification as interpreted in the above document is quoted in the following section of this manual to give information about the usage with OpenTP1:

Chapter 6. X/Open-Compliant Inter-Application Communication (TxRPC)

COBOL

COBOL was developed by CODASYL (the Conference on Data Systems Languages). Of the OpenTP1 application programming interface specifications, the data manipulation language (DML) specification was developed by relying on the communication section in CODASYL COBOL (1981) as well as the RECEIVE, SEND, COMMIT, and ROLLBACK statements and adding original specifications and

interpretations made by Hitachi, Ltd. The publisher of this manual expresses acknowledgment to the original developer and presents the following acknowledgment statement as requested by CODASYL. This statement is quoted from the acknowledgment in the original CODASYL COBOL specification titled COBOL *Journal of Development* 1984.

Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas from this report as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgement paragraphs in their entirety as part of the preface to any such publication. Any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgement of the source, but need not quote the acknowledgement.

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL COBOL Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the Univac (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

Contents

Preface i

Intended readers i
Organization of this manual i
Related publications ii
Conventions: Abbreviations for product names iv
Conventions: Acronyms ix
Conventions: Diagrams xi
Conventions: Differences between JIS and ASCII keyboards xii
Conventions: Differences in installation directory paths xii
Conventions: Fonts and symbols xiii
Conventions: KB, MB, GB, and TB xiv
Conventions: Platform-specific notational differences xiv
Conventions: Version numbers xv
Acknowledgments xv

1. OpenTP1 Application Programs 1

1.1 Relationship between user application programs and communication modes 2
1.1.1 Application programs in client/server mode 3
1.1.2 Application programs in message exchange mode 4
1.1.3 Application programs in message queuing mode 5
1.1.4 Application program load balancing 6
1.1.5 Transaction processing with application program 7
1.2 Types of application program 8
1.2.1 Using services UAP (SUP) 10
1.2.2 Providing services UAP (SPP) 13
1.2.3 Message handling UAP (MHP) 18
1.2.4 UAP that handles offline work 24
1.3 Creation of application programs 26
1.3.1 Coding 28
1.3.2 Creating stubs 31
1.3.3 Compilation and linkage (when using a stub) 34
1.3.4 Compilation and linkage (when using dynamic loading of service functions) 35
1.3.5 Application program environment setup 36
1.3.6 User server load balancing and scheduling 37
1.4 OpenTP1 library functions 46
1.4.1 Application programming interface facilities 46
1.4.2 List of OpenTP1 library functions 47

1.5 Debuggers and testers for application programs ... 69
1.5.1 Types of UAP tester facility ... 69
1.5.2 UAPs that can be tested ... 70
1.5.3 Reporting the test status of a user server ... 70

2. Basic OpenTP1 Facilities (TP1/Server Base, TP1/LiNK) 71

2.1 Remote procedure call ... 72
2.1.1 How to implement the remote procedure call ... 72
2.1.2 Transferring data through the remote procedure call ... 73
2.1.3 Outline of remote procedure call modes ... 74
2.1.4 Nesting services ... 81
2.1.5 Using nontransactional RPC from transaction process ... 82
2.1.6 Setting schedule priorities for service requests ... 82
2.1.7 Acquiring node address of client UAP ... 83
2.1.8 Referencing and changing response waiting intervals of service request ... 83
2.1.9 Acquiring descriptor of asynchronous-response-type RPC request which has encountered error ... 84
2.1.10 Report data to CUP unidirectionally ... 84
2.1.11 Relationship between remote procedure calls and processes for executing services ... 86
2.1.12 Notes on using a recursive call ... 90
2.1.13 Retrying a service function ... 91
2.1.14 User data compression ... 92
2.1.15 Monitoring the service function execution time ... 93
2.1.16 RPC with the multi-scheduler facility ... 94
2.1.17 RPC with a communication destination specified ... 96
2.1.18 Service request with domain qualification ... 98
2.1.19 Relationship between service functions and stubs ... 100
2.2 Remote API facility ... 111
2.2.1 Application of the remote API facility ... 114
2.2.2 Permanent connection ... 116
2.2.3 Connection mode ... 116
2.2.4 Chained RPCs using the remote API facility ... 118
2.2.5 Notes on the remote API facility ... 119
2.3 Transaction control ... 121
2.3.1 Transaction in client/server mode ... 121
2.3.2 Acquiring a synchronization point ... 122
2.3.3 Specification of transaction attribute ... 127
2.3.4 Relationship between remote procedure call modes and synchronization points ... 129
2.3.5 Transaction optimization ... 135
2.3.6 Posting information about the current transaction ... 153
2.3.7 Disposal in case of heuristic situation ... 153
2.3.8 Notes on transaction processing ... 153

2.4 System operation management 155
2.4.1 Executing operation commands 155
2.4.2 Reporting completion of user server start processing 164
2.4.3 Detecting the user server status 165
2.5 Message log output 169
2.5.1 Outputting message log from application programs 169
2.6 Audit log output 173
2.7 User journal acquisition 176
2.8 Journal data editing 178
2.9 Receiving message log notification 180
2.10 Client/server mode communication using OSI TP 182
2.10.1 Application programs used for OSI TP communication 183
2.10.2 SPPs for a communication event 183
2.10.3 Errors encountered during OSI TP communication 185
2.11 Acquiring performance verification traces 186
2.12 Real-time statistical information acquisition 187

3. Facilities Provided by TP1/Message Control 189

3.1 MCF communication service operations 190
3.2 Connection establishment and release 191
3.2.1 Establishing or releasing a connection by issuing a function from the UAP 191
3.2.2 Coding examples for re-establishing or forcibly releasing a connection 194
3.2.3 Start and terminate acceptance of connection establishment requests 198
3.3 Application-related operations 199
3.4 Shutdown and release of logical terminals 200
3.5 Communication protocol products and functions available in operations 202
3.6 Message exchange processing 205
3.6.1 Message communication modes 206
3.6.2 Message structure 215
3.6.3 Receiving messages 216
3.6.4 Sending messages 217
3.6.5 Synchronous message processing 218
3.6.6 Continuous-inquiry-response processing 222
3.6.7 Resending messages 226
3.7 MCF transaction control 228
3.7.1 MHP transaction control 228
3.8 MCF extended facilities 233
3.8.1 Starting application programs 233
3.8.2 MHP startup using command 242
3.8.3 Nontransaction attribute MHP 243
3.8.4 Time monitoring with the facility for user timer monitoring 244
3.9 User exit routines 248

3.9.1 User exit routine that edits input message and application name determination ... 250
3.9.2 User exit routine that determines the inheriting timer-start message ... 251
3.9.3 User exit routine that edits sequential number of send message ... 251
3.9.4 User exit routine that edits output message ... 252
3.10 MCF events ... 253
3.10.1 MCF event that reports detection of an invalid application name (ERREVT1) ... 261
3.10.2 MCF event that reports discarding of a message (ERREVT2) ... 262
3.10.3 MCF event that reports UAP abnormal termination (ERREVT3) ... 263
3.10.4 MCF event that reports discarding of a timer-start message (ERREVT4) 265
3.10.5 MCF event that reports discarding of an unprocessed send message (ERREVTA) ... 266
3.10.6 MCF event that reports a send error (SERREVT) ... 269
3.10.7 MCF event that reports send completion (SCMPEVT) ... 271
3.10.8 MCF event that reports an error (CERREVT, VERREVT) ... 272
3.10.9 MCF event that reports establishing a connection (COPNEVT, VOPNEVT) ... 273
3.10.10 MCF event that reports releasing a connection (CCLSEVT, VCLSEVT) ... 275
3.10.11 Message format for MCF events ... 276
3.11 MCF processes used by application programs ... 279
3.11.1 Types of MCF process ... 281
3.11.2 Files for using MCF processes ... 281

4. Facilities for User Data 285

4.1 DAM file service (TP1/FS/Direct Access) ... 286
4.1.1 DAM file configuration ... 286
4.1.2 Physical files and logical files ... 286
4.1.3 Outline of access to DAM files ... 287
4.1.4 Access to a DAM file in online mode (operation from an SUP, SPP, or MHP) ... 288
4.1.5 Access to a DAM file in offline mode (operation from a UAP that handles offline work) ... 297
4.1.6 Creating physical files (operation from a UAP that handles offline work) 300
4.1.7 Locking DAM files ... 301
4.1.8 Access to unrecoverable DAM files ... 303
4.1.9 Interchangeability of DAM and TAM services ... 312
4.2 TAM file service (TP1/FS/Table Access) ... 313
4.2.1 TAM file configuration ... 313
4.2.2 Conditions for accessing a TAM table ... 314
4.2.3 Name used when a TAM table is accessed ... 315
4.2.4 Procedure for accessing a TAM table ... 315
4.2.5 Relationship between transactions and TAM access ... 319

4.2.6 Lock for TAM tables 329
4.2.7 TAM table access facility without table-based lock 331
4.2.8 Creating TAM files 342
4.2.9 Interchangeability of TAM and DAM services 342
4.2.10 TAM service statistical information 343
4.2.11 Notes on adding and deleting TAM records 343
4.3 IST service (TP1/Shared Table Access) 350
4.3.1 System configuration of IST service 350
4.3.2 Outline of internode shared tables 351
4.3.3 Procedure for accessing an internode shared table 354
4.3.4 Lock for internode shared tables 355
4.4 ISAM file service (ISAM, ISAM/B) 356
4.4.1 Outline of ISAM files 356
4.4.2 Types of ISAM service 356
4.5 Accessing database management systems 358
4.5.1 Relation to OpenTP1 transaction processing 358
4.5.2 Preparation for using other vendors' DBMS in cooperation with OpenTP1 through XA interface 359
4.6 Lock for resources 361
4.6.1 Resources which can be put under lock 361
4.6.2 Types of lock 361
4.6.3 Specifying the maximum lock wait time 362
4.6.4 Insufficient table pool for lock 362
4.6.5 Releasing a resource from lock 362
4.6.6 Lock migration 363
4.6.7 Lock test 364
4.7 Responses to the occurrence of deadlocks 366
4.7.1 Notes for avoiding deadlocks 366
4.7.2 OpenTP1 responses to deadlocks 366

5. X/Open-compliant Application Programming Interface 369

5.1 XATMI interface (client/server-mode communication) 370
5.1.1 Communication paradigms available with XATMI interface 370
5.1.2 XATMI interface functions 371
5.1.3 Request/response service paradigm 375
5.1.4 Conversational service paradigm 379
5.1.5 Notes on using xatmi interface for communication under OpenTP1 383
5.1.6 Communication data types 384
5.1.7 How to create server UAP 389
5.1.8 Relationship between OpenTP1 facility and XATMI interface 390
5.2 TX interface (transaction control) 393
5.2.1 TX interfaces usable with OpenTP1 393
5.2.2 How to use TX_ functions 394
5.2.3 Restrictions on using TX_ functions 396

5.2.4 Comparison with transaction control functions of OpenTP1 (dc_trn_ ~) .. 397

6. X/Open-compliant Inter-application Communication (TxRPC) 399

6.1 Communication through TxRPC interface 400
6.1.1 Types of TxRPC communication..... 400
6.1.2 Application programs that can be created 401
6.1.3 Necessary libraries..... 401
6.2 Communication allowed with application programs 402
6.2.1 TxRPC remote procedure calls..... 402
6.2.2 TxRPC transaction processing..... 402
6.2.3 Relation between application programs using OpenTP1 facilities and TxRPC application programs 403
6.3 Procedures for creating application programs for TxRPC communication..... 404
6.3.1 Procedure for creating UAP for IDL-only TxRPC communication..... 404

7. Facilities Provided by TP1/Multi 407

7.1 Application programs in cluster/parallel mode..... 408
7.1.1 Node on which application programs can be executed 408
7.1.2 Prerequisites to application program execution..... 408
7.2 Facilities available with the use of application programs..... 410
7.2.1 Acquisition of OpenTP1 node status 410
7.2.2 Acquisition of user server status.....411
7.2.3 Acquisition of OpenTP1 node identifier 413
7.3 Conditions for using multinode facility functions 416

8. OpenTP1 Samples 419

8.1 Outline of samples 420
8.1.1 Types of sample programs 420
8.1.2 Sample program directory configuration..... 421
8.1.3 Explanation format of samples 426
8.2 How to use Base sample 428
8.2.1 Procedure common to all samples (Base sample) 429
8.2.2 Tasks specific to the Base sample (when using a stub) 430
8.2.3 Tasks for using OpenTP1 (when using a stub) 433
8.2.4 Tasks specific to the Base sample (when using dynamic loading of service functions)..... 435
8.2.5 Tasks for using OpenTP1 (when using dynamic loading of service functions)..... 437
8.3 How to use DAM sample..... 440
8.3.1 Procedure common to all samples (DAM sample)..... 442
8.3.2 DAM sample specific work 442
8.3.3 Work for using OpenTP1..... 444
8.4 How to use TAM sample 447
8.4.1 Procedure common to all samples (TAM sample) 449

8.4.2 TAM sample specific work 449
8.4.3 Work for using OpenTP1 451
8.5 Specifications of sample programs 455
8.5.1 Contents of database used by samples 455
8.5.2 Outline of sample program processing 455
8.5.3 Structure of sample programs 457
8.5.4 Details of programs specific to each sample 459
8.6 How to use MCF sample 462
8.6.1 MCF sample directory configuration 462
8.6.2 Notes on using MCF sample 466
8.7 Samples to be used to dispatch multi OpenTP1 command 467
8.8 COBOL language templates 469
8.8.1 Files of COBOL language templates 469
8.8.2 How to use the cobol language templates 469
8.9 How to use sample scenario template 472
8.10 How to use real-time acquisition item definition templates 473

Appendixes 475

A. Output Format of Undecided Transaction Information 476
B. Output Format of Deadlock Information 481
C. Examples of System Configurations Requiring Consideration of the Multi-Scheduler Facility 488
C.1 Overview of processing by the scheduler facility 488
C.2 Examples of system configurations in which the scheduler is likely to be the cause of error 491
C.3 Example of a system configuration using the multi-scheduler facility 496
C.4 Notes 504

Index 505

List of figures

Figure 1-1: OpenTP1 and UAP positions in network 3
Figure 1-2: Outlines of UAPs in client/server mode 4
Figure 1-3: Outline of message exchange processing 5
Figure 1-4: Outline of UAPs using Message Queuing 6
Figure 1-5: Outline of UAP roles and positions (client/server mode) 9
Figure 1-6: Outline of UAP roles and positions (message exchange mode) 10
Figure 1-7: Outline of SUP 11
Figure 1-8: Outline of SUP processing (C language) 13
Figure 1-9: Outline of SPP 14
Figure 1-10: SPP configuration (when using a stub) 15
Figure 1-11: SPP configuration (when using dynamic loading of service functions) 16
Figure 1-12: Outline of SPP processing (C language) 18
Figure 1-13: Outline of MHP 19
Figure 1-14: MHP configuration (using a stub) 20
Figure 1-15: MHP configuration (using dynamic loading of service functions) 21
Figure 1-16: Outline of MHP processing (C language) 24
Figure 1-17: Outline of UAP that handles offline work 25
Figure 1-18: Procedure for UAP creation (when using a stub) 27
Figure 1-19: Procedure for UAP creation (when using dynamic loading of service functions) 28
Figure 1-20: Outline of UAP coding in C 30
Figure 1-21: Outline of UAP coding in COBOL 31
Figure 1-22: Stub linked to server UAP 32
Figure 1-23: Stub creation procedure 34
Figure 1-24: Process load balancing 40
Figure 1-25: Outline of internode load-balancing facility 42
Figure 1-26: Example of using the multi-scheduler facility 45
Figure 2-1: Client/server relationship in communication using RPC 73
Figure 2-2: Data transfer through remote procedure call 74
Figure 2-3: RPC modes 74
Figure 2-4: Synchronous-response-type RPC 76
Figure 2-5: Asynchronous-response-type RPC (asynchronous receiving of processing results) 78
Figure 2-6: Asynchronous-response-type RPC (rejection of receiving processing results) 80
Figure 2-7: Nonresponse-type RPC 81
Figure 2-8: Example of nesting RPCs 82
Figure 2-9: Outline of reporting data to CUP unidirectionally 85
Figure 2-10: Relationship between RPCs and processes 86
Figure 2-11: Relationship between chained RPCs and processes 89
Figure 2-12: Outline of retry of service function 92

Figure 2-13: Outline of the data compression facility 93
Figure 2-14: Outline of service function execution time monitoring 94
Figure 2-15: Outline of an RPC with the multi-scheduler facility 95
Figure 2-16: Example of communication using the function dc_rpc_call_to() 98
Figure 2-17: Outline of service request with domain qualification 100
Figure 2-18: Using a stub to acquire service functions (SPP) 102
Figure 2-19: Using a stub to acquire service functions (MHP) 104
Figure 2-20: Using dynamic loading of service functions only (SPP) 106
Figure 2-21: Using dynamic loading of service functions only (MHP) 107
Figure 2-22: Using both dynamic loading of service functions and a stub (SPP) 109
Figure 2-23: Using both dynamic loading of service functions and a stub (MHP) 110
Figure 2-24: Remote API facility 112
Figure 2-25: Remote procedure call to a UAP within a firewall 115
Figure 2-26: Outline of automatic connection mode 117
Figure 2-27: Outline of non-automatic connection mode 118
Figure 2-28: Transactions in chained/unchained mode 123
Figure 2-29: Transaction rollback 125
Figure 2-30: Transaction rollback if an error occurs during synchronization point acquisition processing 126
Figure 2-31: Relationship between RPCs and transaction attribute 128
Figure 2-32: Relationship between synchronous-response-type RPC and synchronization point 130
Figure 2-33: Relationship between asynchronous-response-type RPC and synchronization point 131
Figure 2-34: Relationship between nonresponse-type RPC and synchronization point 132
Figure 2-35: Relationship between chained RPCs and synchronization point (transactional chained RPCs) 133
Figure 2-36: Relationship between chained RPCs and synchronization points (if a specification is given so that the server processing will not end with the non-transactional chained RPCs) 134
Figure 2-37: Outline of ordinary transaction processing (two-phase commit) 136
Figure 2-38: Outline of commit optimization 138
Figure 2-39: Outline of prepare optimization 140
Figure 2-40: Outline of asynchronous prepare optimization 142
Figure 2-41: Outline of one-phase optimization 144
Figure 2-42: Outline of read-only optimization 146
Figure 2-43: Outline of no-access optimization 148
Figure 2-44: Outline of rollback optimization 150
Figure 2-45: Outline of optimization using chained RPCs 152
Figure 2-46: Outline of OpenTP1 command execution using function dc_adm_call_command () 155
Figure 2-47: Transition of user server status (SUP) 166
Figure 2-48: Transition of user server status (SPP, MHP) 167

Figure 2-49: Transition of user server status (server that receives requests from socket (SPP)) 168
Figure 2-50: Outline of message log output from UAP 169
Figure 2-51: Output format of message logs 171
Figure 2-52: Outline of audit logging from UAPs 173
Figure 2-53: Acquiring user journals 177
Figure 2-54: Journal data editing 179
Figure 2-55: Reception of message log notification 181
Figure 2-56: Concept of client/server mode communication using OSI TP 182
Figure 2-57: Outline of SPP for a communication event 184
Figure 2-58: Example of acquiring real-time statistical information in arbitrary sections 188
Figure 3-1: Example of establishing a connection using the function `dc_mcf_tactcn()` 192
Figure 3-2: Example of releasing a connection using the function `dc_mcf_tdctcn()` 193
Figure 3-3: UAP example for automatically re-establishing a connection 194
Figure 3-4: UAP example for forcibly releasing a connection 196
Figure 3-5: Outline of message exchange mode communication 206
Figure 3-6: Message communication modes 207
Figure 3-7: Relationship between logical message and segments 216
Figure 3-8: Message receiving 217
Figure 3-9: Message send processing (asynchronous message sending) 218
Figure 3-10: Synchronous message processing 221
Figure 3-11: Outline of continuous-inquiry-response processing 225
Figure 3-12: Relationship between message exchange processing and transactions 231
Figure 3-13: How to start application program 236
Figure 3-14: Starting MHP from MHP that received send-only message 238
Figure 3-15: Starting MHP from MHP that received inquiry-response message 239
Figure 3-16: Starting MHP, which sends send-only message, from MHP that handles inquiry-response message processing 240
Figure 3-17: Starting MHP from SPP handling transaction processing 241
Figure 3-18: MHP activation by operation command 243
Figure 3-19: Example of using the facility for user timer monitoring 246
Figure 3-20: Positions of user exit routines 249
Figure 3-21: Outline of ERREVT1 261
Figure 3-22: Outline of ERREVT2 263
Figure 3-23: Outline of ERREVT3 264
Figure 3-24: Outline of ERREVT4 266
Figure 3-25: Outline of ERREVTA 268
Figure 3-26: Outline of SERREVT 270
Figure 3-27: Outline of SERREVT 271
Figure 3-28: Outline of CERREVT (VERREVT) 273
Figure 3-29: Outline of COPNEVT (VOPNEVT) 274
Figure 3-30: Outline of CCLSEVT (VCLSEVT) 275
Figure 3-31: Segments of logical message passed as MCF event 277
Figure 3-32: Outline of MCF processes used by UAPs 280

Figure 3-33: Configuration of directories for storing files needed to use MCF service.....283
Figure 4-1: DAM file configuration 286
Figure 4-2: Access to DAM files in online mode.....290
Figure 4-3: Procedure for accessing DAM files in offline mode.....300
Figure 4-4: Procedure for creating DAM file 301
Figure 4-5: Procedure for accessing unrecoverable DAM file 305
Figure 4-6: TAM file configuration 314
Figure 4-7: Access to TAM tables 318
Figure 4-8: Locking resources when updating records.....333
Figure 4-9: Locking resources when adding records.....334
Figure 4-10: Processing that is performed by the TAM table access facility with table-based lock when competition for access to the same record occurs 335
Figure 4-11: Processing that is performed by the TAM table access facility without table-based lock when competition for access to the same record occurs 336
Figure 4-12: How the dc_tam_open function locks resources 337
Figure 4-13: An example of a DCTAMER_NOAREA error caused by an attempt to add records.....338
Figure 4-14: Processing that is performed to obtain a number of empty records equal to the number of records to be added.....339
Figure 4-15: Processing that is performed to add records after the record deletion transaction is committed.....340
Figure 4-16: Example in which a deadlock occurs after change from a TAM table that uses the TAM table access facility with table-based lock to a TAM table that uses the TAM table access facility without table-based lock 345
Figure 4-17: Example of update and addition 347
Figure 4-18: Occurrence of a deadlock.....348
Figure 4-19: When the same value is not specified in the internode shared table definition for all nodes.....351
Figure 4-20: Updating an internode shared table record 352
Figure 4-21: Procedures for accessing internode shared tables.....355
Figure 4-22: Form of ISAM file services 357
Figure 4-23: Outline of lock migration.....364
Figure 5-1: Communication with synchronous response reception based on request/response service paradigm 376
Figure 5-2: Communication with asynchronous response reception based on request/response service paradigm 377
Figure 5-3: Communication without response reception based on request/response service paradigm.....378
Figure 5-4: Communication based on conversational service paradigm.....382
Figure 6-1: Outline of communication through TxRPC interface.....400
Figure 6-2: Communication by using application programs 403
Figure 6-3: Procedure for creating UAP for IDL-only TxRPC communication 405
Figure 7-1: Outline of application programs in cluster/parallel mode.....409
Figure 7-2: Procedure for acquiring OpenTP1 nodes in succession.....411

Figure 7-3: Procedure for acquiring user server statuses in succession 413
Figure 7-4: Procedure for acquiring OpenTP1 node identifiers in succession 414
Figure 8-1: Configuration of directories for storing samples 422
Figure 8-2: Outline of procedure for using samples (Base sample when using a stub) 428
Figure 8-3: Outline of procedure for using samples (Base sample when using dynamic loading of service functions) 429
Figure 8-4: Outline of procedure for using samples (DAM sample) 441
Figure 8-5: Outline of procedure for using samples (TAM sample) 448
Figure 8-6: Relationship between client and server UAP calls (C language) 456
Figure 8-7: Relationship between client and server UAP calls (COBOL language) 457
Figure 8-8: Program structure of client and server UAPs (C language) 458
Figure 8-9: Program structure of client and server UAPs (COBOL language) 459
Figure 8-10: Program structure of client and server UAPs (DAM sample written in COBOL language) 460
Figure 8-11: Configuration of directories for MCF sample 463
Figure A-1: Output format of undecided transaction information 477
Figure A-2: Output example of undecided transaction information 480
Figure B-1: Output format of deadlock information 482
Figure B-2: Output example of deadlock information 483
Figure B-3: Output format of timeout information 484
Figure B-4: Output example of timeout information 486
Figure B-5: Output format of TAM resource deadlock information 487
Figure C-1: Overview of processing by the scheduler facility 489
Figure C-2: Overview of processing service request messages 490
Figure C-3: Example of a system that has insufficient socket descriptors 492
Figure C-4: Example of a system in which the connect system call encountered an error 493
Figure C-5: Example of a system using networks that have different line speeds 494
Figure C-6: Example of a system in which service request messages are interrupted 495
Figure C-7: Example of a system in which the processing threads are temporarily deficient 496
Figure C-8: Example of a system configuration that solves the deficiency of socket descriptors 497
Figure C-9: Example of a system configuration that solves errors with the connect system call 498
Figure C-10: Example of a system that effectively uses a network having a high line speed 500
Figure C-11: Example of a system in which service request messages are not interrupted 502
Figure C-12: Example of a system with an increased number of simultaneously executable processing threads 503

List of tables

Table 1-1: OpenTP1 library functions (basic OpenTP1 facilities) 48
Table 1-2: OpenTP1 library functions (TP1/Message Control functions)....50
Table 1-3: OpenTP1 library functions (user data manipulation functions) 53
Table 1-4: OpenTP1 library functions (X/Open-compatible functions)....55
Table 1-5: OpenTP1 library functions (functions used in special style)....57
Table 1-6: Library functions available with UAPs (basic OpenTP1 facilities) 58
Table 1-7: Library functions available with UAPs (TP1/Message Control functions)....61
Table 1-8: Library functions available with UAPs (operate user data) 63
Table 1-9: Library functions available with UAPs (X/Open-compatible functions)....65
Table 1-10: Library functions available with UAPs (functions used in special style)....67
Table 2-1: OpenTP1 commands which can be executed from UAPs....156
Table 2-2: Contents of message logs output to message log file 170
Table 2-3: Items output to audit log file....174
Table 2-4: Specifying the flags argument to the function dc_rts_utrace_put()....187
Table 3-1: Functional differences between the function and the operation command (MCF communication service operations)....190
Table 3-2: Functional differences between functions and operation commands (connection establishment and release)....193
Table 3-3: Functional differences between functions and operation commands (start and terminate acceptance of connection establishment requests)....198
Table 3-4: Functional differences between the function and operation command (application-related operations)....199
Table 3-5: Functional differences between functions and operation commands (shutdown and release of logical terminals) 201
Table 3-6: Communication protocol products and functions available in operations (1/3)....202
Table 3-7: Communication protocol products and functions available in operations (2/3)....203
Table 3-8: Communication protocol products and functions available in operations (3/3)....203
Table 3-9: Correspondence between the types of application and message exchange functions....209
Table 3-10: Functions available in communication modes used by communication protocol products (1/5) 210
Table 3-11: Functions available in communication modes used by communication protocol products (2/5) 211
Table 3-12: Functions available in communication modes used by communication protocol products (3/5) 212
Table 3-13: Functions available in communication modes used by communication protocol products (4/5) 213
Table 3-14: Functions available in communication modes used by communication protocol products (5/5) 214
Table 3-15: User exit routines available with OpenTP1....250

Table 3-16: MCF events 253
Table 3-17: Relationship between MHPs for an MCF event and application attributes 256
Table 3-18: Relationship between communication protocol products and reported MCF events (1/5) 256
Table 3-19: Relationship between communication protocol products and reported MCF events (2/5) 257
Table 3-20: Relationship between communication protocol products and reported MCF events (3/5) 258
Table 3-21: Relationship between communication protocol products and reported MCF events (4/5) 259
Table 3-22: Relationship between communication protocol products and reported MCF events (5/5) 260
Table 4-1: Functions able to access the same block in one transaction (recoverable DAM files) 291
Table 4-2: Functions able to access the same block in different transaction (recoverable DAM files) 294
Table 4-3: Functions able to access the same block in one UAP (unrecoverable DAM files) 305
Table 4-4: Functions able to access the same block in different UAP (unrecoverable DAM files) 307
Table 4-5: Differences in access to recoverable and unrecoverable DAM files 310
Table 4-6: Differences in locking range upon access to recoverable and unrecoverable DAM files 311
Table 4-7: Processing results when function was called more than once for the same record (in one global transaction) 320
Table 4-8: Processing results when function was called more than once for the same record (in a different global transaction) 324
Table 4-9: Lock specifications in TAM service functions and actual lock statuses 330
Table 4-10: Actual lock status, as compared to the lock setting of the TAM service function used to activate the TAM table access facility without table-based lock 332
Table 4-11: Conditions that require program recompilation 341
Table 4-12: Conditions that require program relinkage 342
Table 4-13: Combinations of lock modes and resource sharing enabled/disabled 362
Table 5-1: XATMI interface library functions 371
Table 5-2: Relationship between XATMI interface functions and OpenTP1 UAPs 372
Table 5-3: Relationship between XATMI interface facilities and communication protocol .. 374
Table 5-4: Data types that can be used with each communication data type 386
Table 5-5: Relationship between online tester facilities and XATMI interface 391
Table 5-6: TX_ functions available with OpenTP1 UAPs 393
Table 5-7: Relationship between OpenTP1 UAPs and TX_ functions 394
Table 5-8: Relationship between TX_ functions and transaction control functions of OpenTP1 (dc_trn_ ~) 397
Table 7-1: Conditions for using multinode facility functions 416
Table 8-1: Definition files and content to be modified (Base sample) 431

Table 8-2: List of files in OpenTP1 file system (Base sample) 434
Table 8-3: Definition files and content to be modified (Base sample) 436
Table 8-4: List of files in the OpenTP1 file system (Base sample) 438
Table 8-5: Definition files and content to be modified (DAM sample) 443
Table 8-6: List of files in OpenTP1 file system (DAM sample) 446
Table 8-7: Definition files and content to be modified (TAM sample) 450
Table 8-8: List of files in OpenTP1 file system (TAM sample) 453
Table 8-9: Specifications of the TAM sample file 453
Table 8-10: Format of customer information database 455
Table 8-11: File name and content of each real-time acquisition item definition file 473

Chapter

1. OpenTP1 Application Programs

This chapter outlines OpenTP1 application programs.

The facilities are explained using C-language function names. For each function, the name of the equivalent COBOL-language API function is indicated in brackets [ ] when the function appears first in this chapter. After that, only the C-language function name is written.

This chapter contains the following sections:

1.1 Relationship between user application programs and communication modes
1.2 Types of application program
1.3 Creation of application programs
1.4 OpenTP1 library functions
1.5 Debuggers and testers for application programs

1.1 Relationship between user application programs and communication modes

OpenTP1# application programs (UAP: User Application Program) are created to perform online transaction processing for communication among the mainframe, workstations (WSs), personal computers (PCs), and distributors connected through a network (LAN or WAN).

Three communication modes are available with OpenTP1 UAPs:

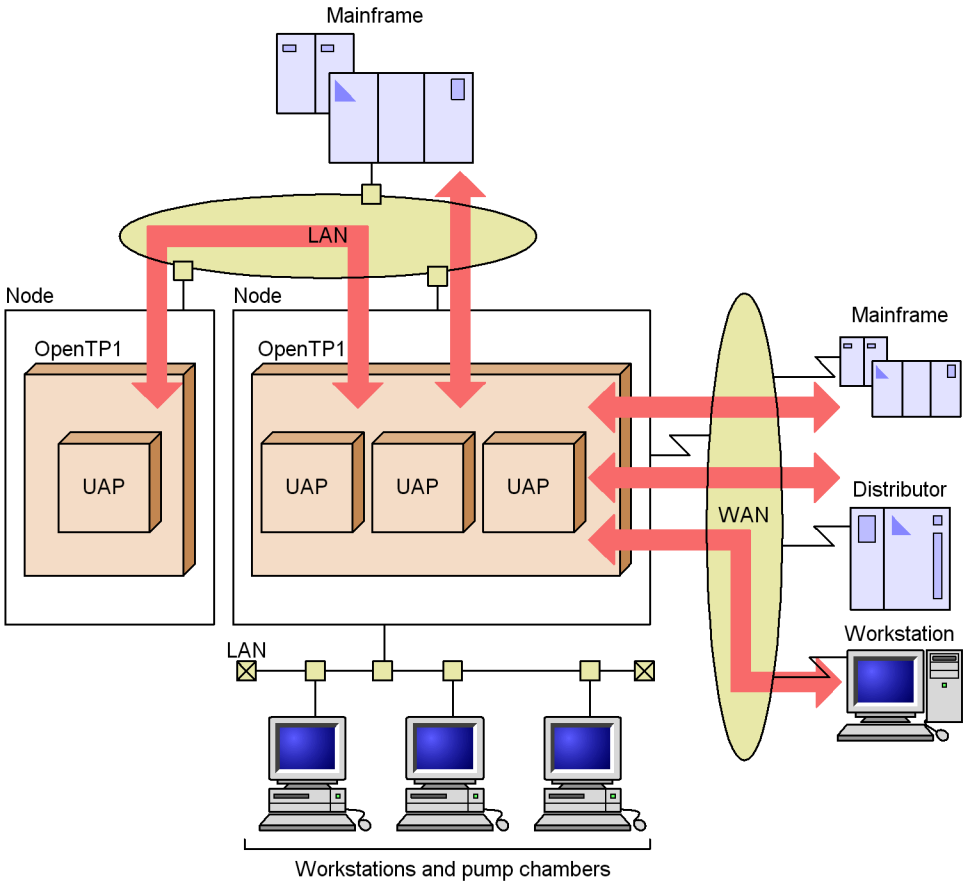
- Client/server mode UAP
- Message exchange mode UAP
- Message Queuing mode UAP

#

Throughout this manual, both the distributed transaction processing facility TP1/Server Base and the distributed application server TP1/LiNK are referred to as *OpenTP1*.

The figure below shows the positions of OpenTP1 and UAPs in the network.

Figure 1-1: OpenTP1 and UAP positions in network



1.1.1 Application programs in client/server mode

A UAP in client/server mode can call and use the program of another process. Units of programs which can be called and used are called *services*. Processes which provide services are called *servers*. UAP servers are called *user servers*.

UAPs in client/server mode are categorized into two types: UAPs (client UAPs) which request services and UAPs (server UAPs) which provide services. A client UAP and a server UAP are required to implement one job.

A server UAP must be created only to provide services. The server UAP can be shared by multiple UAPs.

A client UAP can request a server UAP service by using a remote procedure call (*RPC*). The RPC allows the client UAP to request the server without recognizing the node at which the server UAP exists. Also, the UAPs do not have to consider the

communication protocol between nodes.

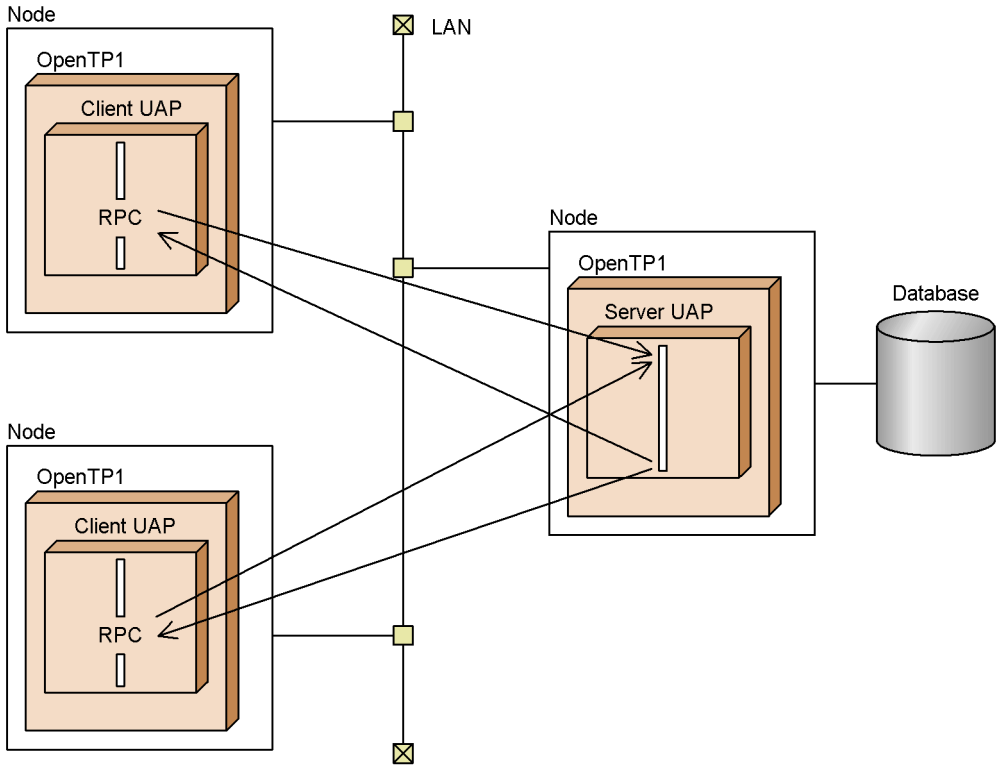
For client/server mode communication under the OpenTP1, either TCP/IP or OSI TP can be used as the communication protocol. For either case, UAPs need not be aware of the internode communication protocol.

Node

In this manual, a node means a machine at which OpenTP1 connected to a network operates. When multi OpenTP1 is used, a node consists of more than one OpenTP1.

The figure below shows UAPs in client/server mode.

Figure 1-2: Outlines of UAPs in client/server mode



1.1.2 Application programs in message exchange mode

UAPs in message exchange mode enable communication between a host and a distributor which are connected through a protocol (e.g., OSI TP) that complies with the OSI, TCP/IP, and conventional networks.

The host and the distributor communicate with each other by sending and receiving

messages. UAPs in message exchange mode are mainly used for communication with an own system which is in a wide area network (WAN) via a communication management program.

Coding formats are different between a UAP for message exchange processing and a UAP using the RPC in client/server mode. Unlike UAPs used for processing in client/server mode, once a UAP is created for message exchange processing, this UAP is used only to send and receive messages.

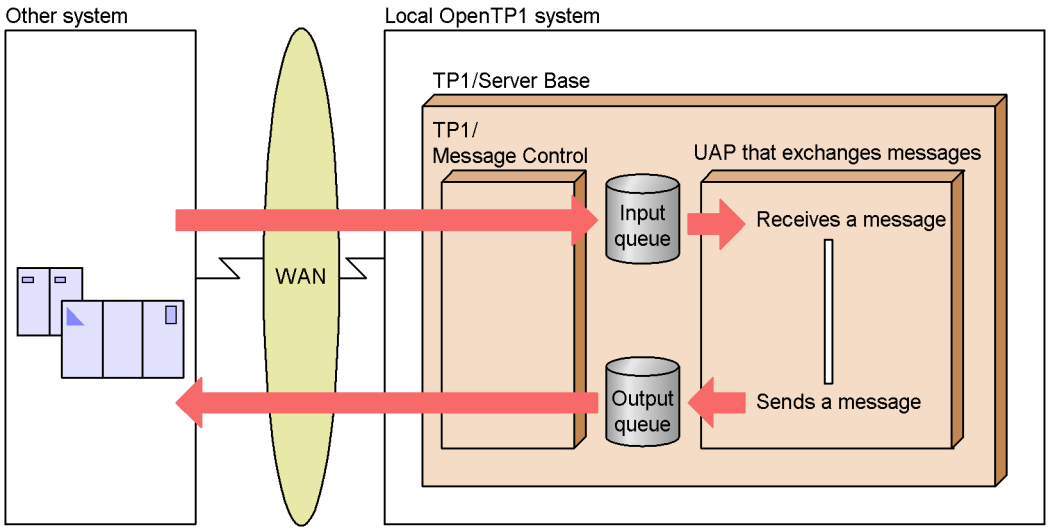
Nodes that use a UAP for message exchange processing must incorporate the OpenTP1 message exchange facility (TP1/Message Control, TP1/NET/Library), as well as products that support the appropriate communication protocol (TP1/NET/*xxx*)[#].

#

Throughout this manual, the OpenTP1 message exchange facility (TP1/Message Control, TP1/NET/Library) and products supporting the appropriate communication protocol are referred to as *Message Control Facility* (*MCF*) or the *MCF service*.

The figure below shows how messages are sent and received in an OpenTP1 system.

Figure 1-3: Outline of message exchange processing



1.1.3 Application programs in message queuing mode

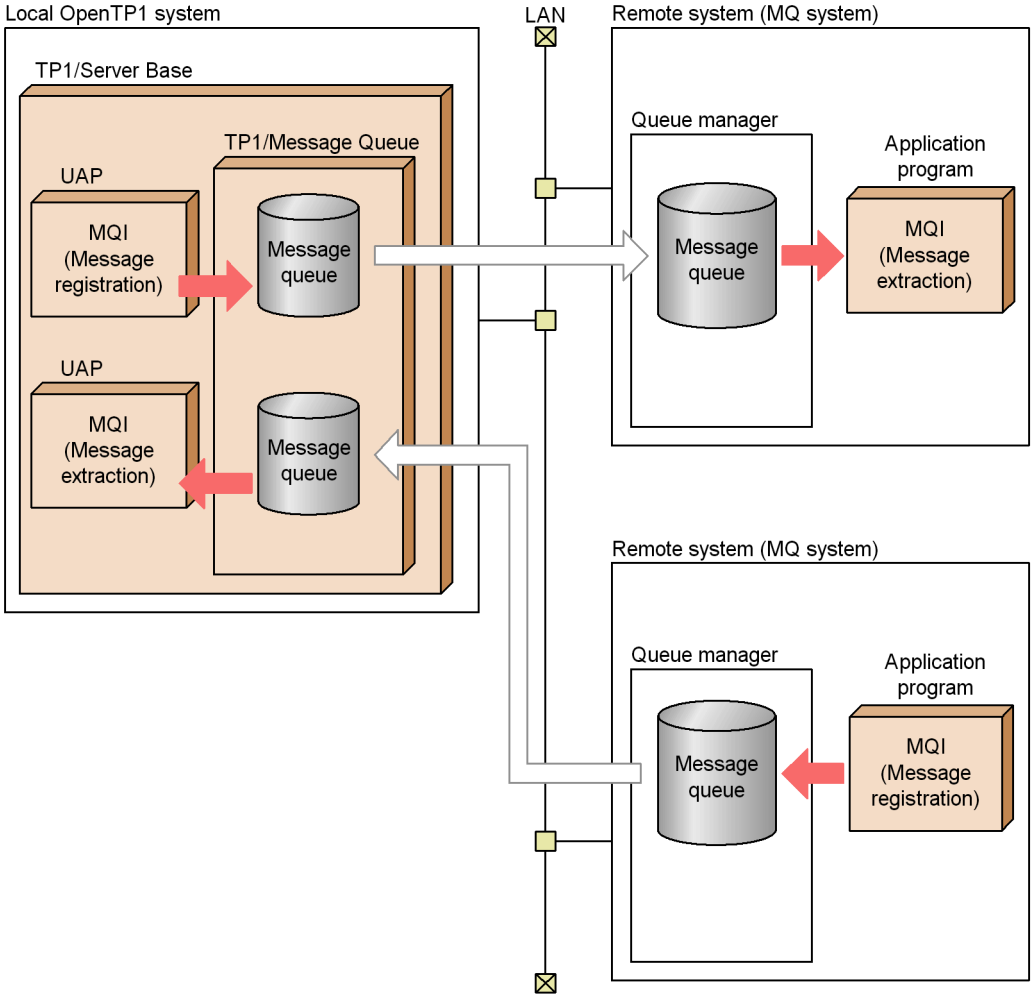
A UAP in Message Queuing mode communicates by putting and getting elements of a data storing queue (message queue). Like electronic mail, data can be sent and received even if the remote system's application program is not running.

Use the API called *Message Queue Interface* (*MQI*) from a UAP to use Message Queuing.

OpenTP1 nodes which use Message Queuing must have TP1/Message Queue. For the usage of Message Queuing, see the *OpenTP1 TP1/Message Queue User's Guide*.

The figure below shows UAPs using Message Queuing.

Figure 1-4: Outline of UAPs using Message Queuing



1.1.4 Application program load balancing

OpenTP1 enables jobs to be executed efficiently by running UAPs in multiple processes. On a node, one UAP processing is executed in multiple processes to

increase efficiency of the server system. This facility is called *multiserver*. It also enables UAPs having the same name to be placed on multiple machines so that service requests can be handled on any node. This facility is called *internode load-balancing facility*. For details of the relationship between UAPs and processes to be executed, see *1.3.6 User server load balancing and scheduling*.

1.1.5 Transaction processing with application program

UAP processing must be divided into units of job processing in order to determine whether to enable/disable the results of each processing. Units in which processing is either enabled/disabled are called *transactions*. OpenTP1 UAPs ensure processing in these transactions.

A division for each job processing in transactions is called a *synchronization point*. When transaction processing reaches the synchronization point, a decision is made on whether the transaction processing terminated normally (enabled) or abnormally (disabled). Acquisition of a synchronization point at which processing terminated normally is called *commitment*. If transaction processing fails to terminate normally without reaching a synchronization point, OpenTP1 cancels processing up to the abnormal termination and recovers on the assumption that the processing did not exist. This synchronization point processing is called *rollback* (partial recovery).

(1) Transaction processing with UAP in client/server mode

OpenTP1 can execute, as transactions, processing of UAPs which use the RPC in client/server mode. Processing which continues requesting many services extending over different nodes can also be treated as one transaction processing.

Transaction processing of UAP in client/server processing can be executed when the UAP calls functions which specify transaction start and commitment. Multiple services nested by the UAP that declared the transaction start can be processed as one transaction.

OpenTP1 enables UAPs in client/server mode to maintain the reliability of transaction processing in conventional data communication.

(2) Transaction processing in message exchange mode

A message handling UAP can be processed as a transaction from the start to the end of message processing. In this case, OpenTP1 automatically controls synchronization point processing.

After the message handling UAP receives a message, transaction control functions for processing in client/server mode cannot be used.

1.2 Types of application program

OpenTP1 UAPs are available in the following types:

- UAPs used for communication in client/server mode
  - Service using program (SUP)

    This UAP is dedicated to a client. The SUP requires the basic OpenTP1 facility (TP1/Server Base or TP1/LiNK).
  - Service providing program (SPP)

    This UAP (server UAP) offers service upon request from a client UAP. The SPP requires the basic OpenTP1 facility (TP1/Server Base or TP1/LiNK).
- UAPs used for communication in message exchange mode
  - Message handling program (MHP)

    This UAP receives and processes messages sent through communication lines. An MHP process can request an SPP for service. The MHP requires the basic OpenTP1 facility and message exchange facility (TP1/Message Control).
- UAPs which initialize user files
  - UAP that handles offline work

    This UAP performs user-specified arbitrary processing. UAPs that handle offline work can use only OpenTP1 library functions for initially creating DAM files and accessing them in a batch environment.
- UAPs used with the OpenTP1 client facility (TP1/Client)
  - Client user program (CUP)

    This UAP is dedicated to a client. Programs which request SPPs for service using TP1/Client library functions from WSs or PCs are called by the generic name of CUP. The CUP requires the OpenTP1 client facility (TP1/Client/W or TP1/Client/P).

    For the CUP, see the manual *OpenTP1 TP1/Client User's Guide TP1/Client/W, TP1/Client/P*.

    You can use TP1/Client for Java to create Java applets and Java applications that request services from SPPs.

    You can use TP1/Client/J to create Java applets, Java applications, and Java servlets. For further information, see the manual *OpenTP1 TP1/Client User's Guide TP1/Client/J*.

Figure 1-5 shows client/server mode UAPs (SUP, SPP) and Figure 1-6 shows a message exchange mode UAP (MHP).

Figure 1-5: Outline of UAP roles and positions (client/server mode)

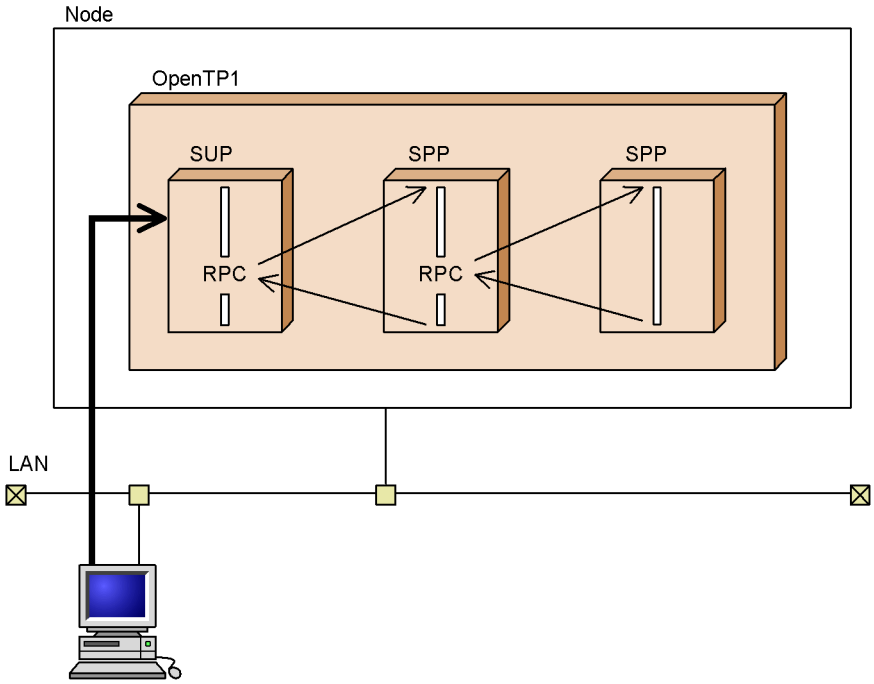
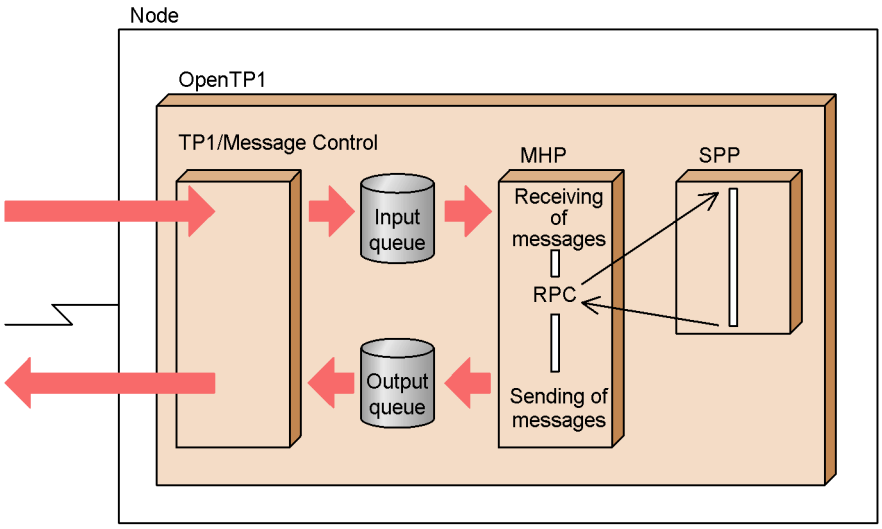


Figure 1-6: Outline of UAP roles and positions (message exchange mode)



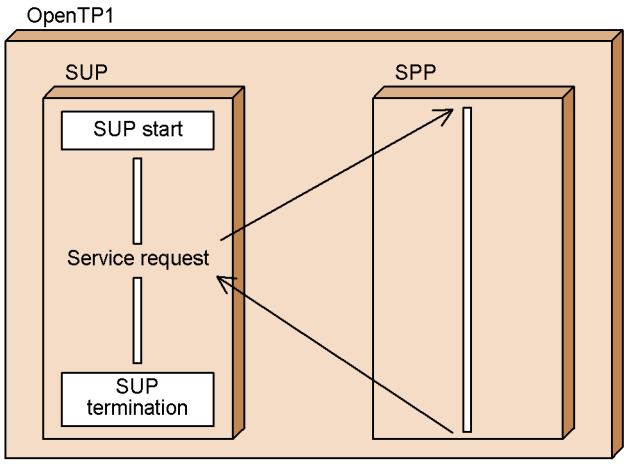
1.2.1 Using services UAP (SUP)

A UAP dedicated to a client is called a *service using program* (*SUP*). An SUP is a UAP which requests the server UAP (SPP) for service and starts communication in client/server mode.

Communication started by an SUP is used only for requesting the SPP for service. It is impossible to create functions for making other UAPs offer service.

The figure below shows the outline of the SUP.

Figure 1-7: Outline of SUP



(1) SUP start

The SUP can be started either at the same time as OpenTP1 or anytime after OpenTP1 is started. If the first method is selected, UAP processing will start as soon as OpenTP1 starts. The starting time can be selected according to the purpose of the created SUP.

(a) Starting at the same time as OpenTP1

Before starting OpenTP1, specify that the SUP is to start at the same time as OpenTP1. The specification method is as follows:

- TP1/Server Base

  Specify the user server name of the SUP in the definition command `dcsvstart` for the user service configuration definition.

- TP1/LiNK

  When setting up the user server environment, specify that the SUP is to start automatically.

(b) Starting anytime after OpenTP1 is started

To start the SUP after OpenTP1 is started, specify the user server name of the SUP as the argument to the `dcsvstart` command.

(2) During SUP operation

Reserve the SUP process as a resident process.

If an error occurs during the SUP process in online mode, the SUP process can be started automatically from another process. In the case of TP1/Server Base, to make another process start the SUP process, specify `Y` for `auto_restart` in the user service

definition. In the case of TP1/LiNK, automatic startup of the SUP process is specified.

If automatic restart is impossible under OpenTP1, use the `dcsvstart` command.

(3) SUP termination

SUP termination is not under OpenTP1 control. If the SUP is to be normally terminated when the intended work terminates, design the SUP so that it will terminate itself. If the SUP requests to start a transaction, it must be terminated after commitment the transaction (acquire the synchronization point). If it is necessary to bring the SUP to abnormal termination when processing is unsuccessful, design the SUP so that it will terminate itself by using `exit()` or `abort()`.

The SUP cannot be normally terminated by the `dcsvstop` command. However, the SUP can be forced to termination using the `dcsvstop -f` command.

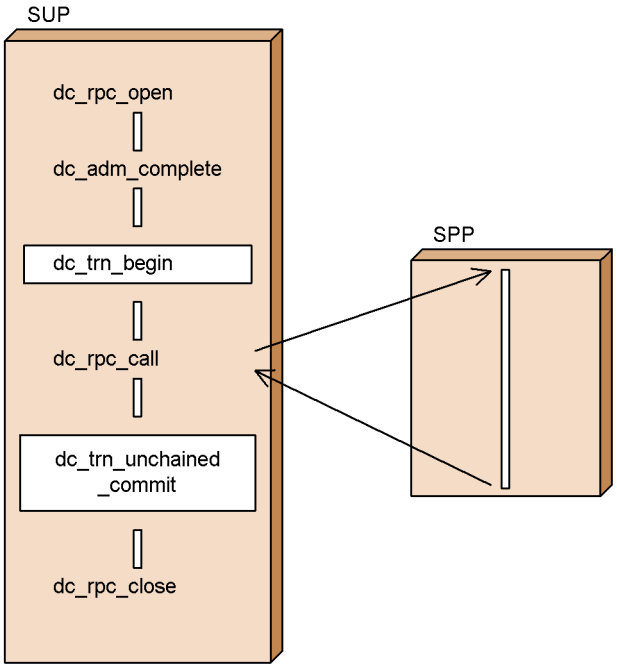
Do not use the `kill` command to terminate the SUP process.

(4) Outline of SUP processing

After using the UAP start function (`dc_rpc_open()`[`CBLDCRPC('OPEN  ')`]) with the SUP, call the user server start completion report function (`dc_adm_complete()`[`CBLDCADM('COMPLETE')`]) to post the completion of server start to OpenTP1.

The figure below shows SUP processing.

Figure 1-8: Outline of SUP processing (C language)



1.2.2 Providing services UAP (SPP)

A UAP which offers a requested service is called a *service providing program* (*SPP*). While OpenTP1 is active, the SPP offers the service requested by the client UAP. The client UAP requests the SPP for service in a way similar to a function call. The client UAP need not be aware which node the SPP exists at.

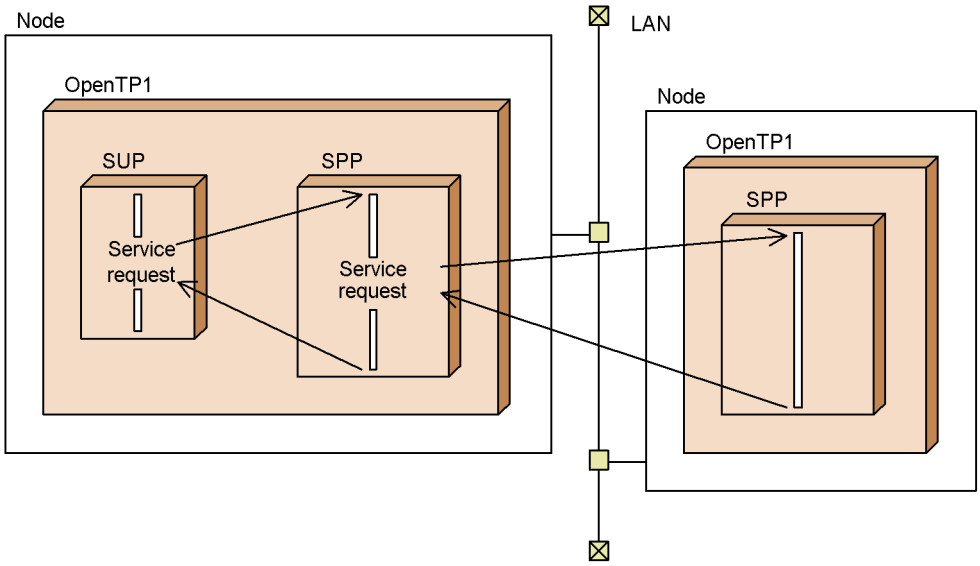
The SPP starts its service when requested. It waits for a request while it is not offering service.

The SPP works as a server by accessing a user file at the node containing OpenTP1. The SPP can access OpenTP1-specific files by way of library functions and can access ORACLE and other DBMS via SQL statements.

An SPP can request another SPP for service, meaning that services can be nested.

The figure below shows an outline of an SPP.

Figure 1-9: Outline of SPP



(1) SPP configuration

Multiple services corresponding to requests of various client UAPs are created. The created services are grouped into an SPP executable file. When C language is used, each service is called a *service function*; when COBOL language is used, each service is called a *service program*. To produce an SPP executable file, link multiple services with a main function (or a main program in COBOL language). Then, define the SPP executable file, comprising one main function and multiple service functions, as a service group in OpenTP1.

The facility for dynamic loading of service functions allows multiple services to be rolled into a UAP shared library#. This eliminates the need to link the services with the main function.

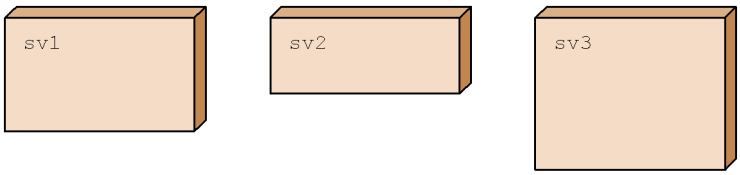
#

Refers to the concept of compiling UAP source files to produce UAP object files, which are then linked to create a shared library.

The figures below show SPP configurations, the first of which uses a stub and the second of which uses dynamic loading of service functions.

Figure 1-10: SPP configuration (when using a stub)

1. A service to be provided to the client UAP is created as a service function.



2. When a main function is created, compiled, and linked, an SPP executable file is created.

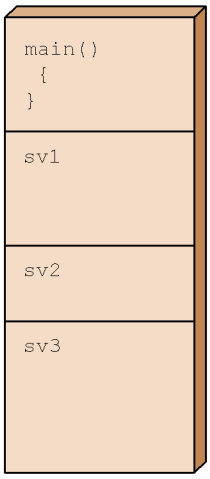
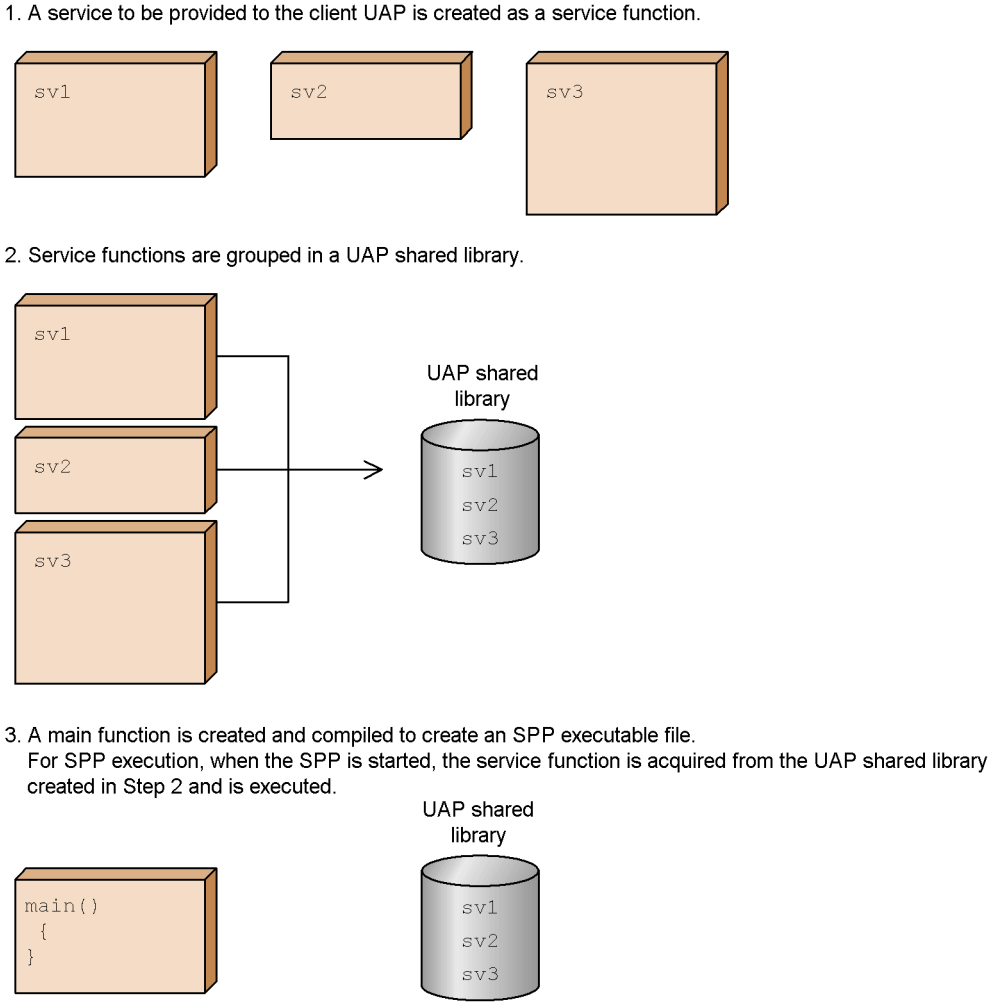


Figure 1-11: SPP configuration (when using dynamic loading of service functions)



(2) SPP start

The SPP can be started either at the same time as OpenTP1 or anytime after OpenTP1 is started. If the first method is selected, SPP processing will start as soon as OpenTP1 starts. The starting time can be selected according to the purpose of the SPP.

(a) Starting at the same time as OpenTP1

Before starting OpenTP1, specify that the SPP is to start at the same time as OpenTP1.

The specification method is as follows:

- TP1/Server Base

  Specify the user server name of the SPP in the definition command `dcsvstart` for the user service configuration definition.

- TP1/LiNK

  During the operation to set up the user server environment, specify that the SPP is to start automatically.

(b) Starting anytime after OpenTP1 is started

To start the SPP anytime after OpenTP1 is started, specify the user server name of the SPP as the argument to the `dcsvstart` command.

The SPP process is initiated from the main function. It becomes ready to offer service when the SPP service starting function (`dc_rpc_mainloop()`[`CBLDCRSV('MAINLOOP')`]) is normally executed.

(3) During SPP operation

The started SPP works as a previously specified process so that memory can be used efficiently. The started SPP may be activated as a resident process or nonresident process. In the former case, the SPP starts processing upon receiving a service request. Even in the latter case, a service request activates the process, thereby starting SPP processing.

For details about setting up UAP processes, see *1.3.5 Application program environment setup*.

(4) SPP termination

The SPP is normally terminated when:

- OpenTP1 is normally terminated.
- The `dcsvstop` command in which the user server name of the SPP is specified is executed during OpenTP1 operation.

When one of the above events occurs, the function `dc_rpc_mainloop()` returns, thereby terminating the SPP.

Do not use the `kill` command to terminate the SPP process.

(5) Outline of SPP processing

Perform the following pre-processing for SPP main functions:

- Application program start (`dc_rpc_open()`[`CBLDCRPC('OPEN  ')`]).
- SPP service start (`dc_rpc_mainloop()`[`CBLDCRSV('MAINLOOP')`]).

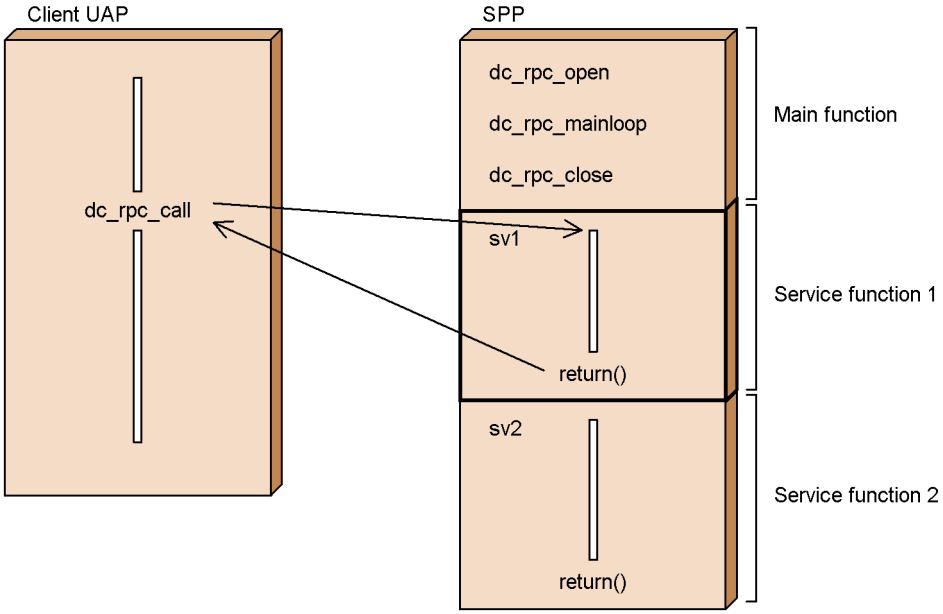
If the transaction start function has been called from the SPP, terminate the SPP after

using the transaction commitment function (synchronization point acquisition).

To use an MCF function from the SPP, call the MCF environment open function (`dc_mcf_open()[CBLDCMCF('OPEN  ')]`) and the MCF environment close function (`dc_mcf_close()[CBLDCMCF('CLOSE  ')]`) as main functions.

The figure below shows SPP processing.

Figure 1-12: Outline of SPP processing (C language)



1.2.3 Message handling UAP (MHP)

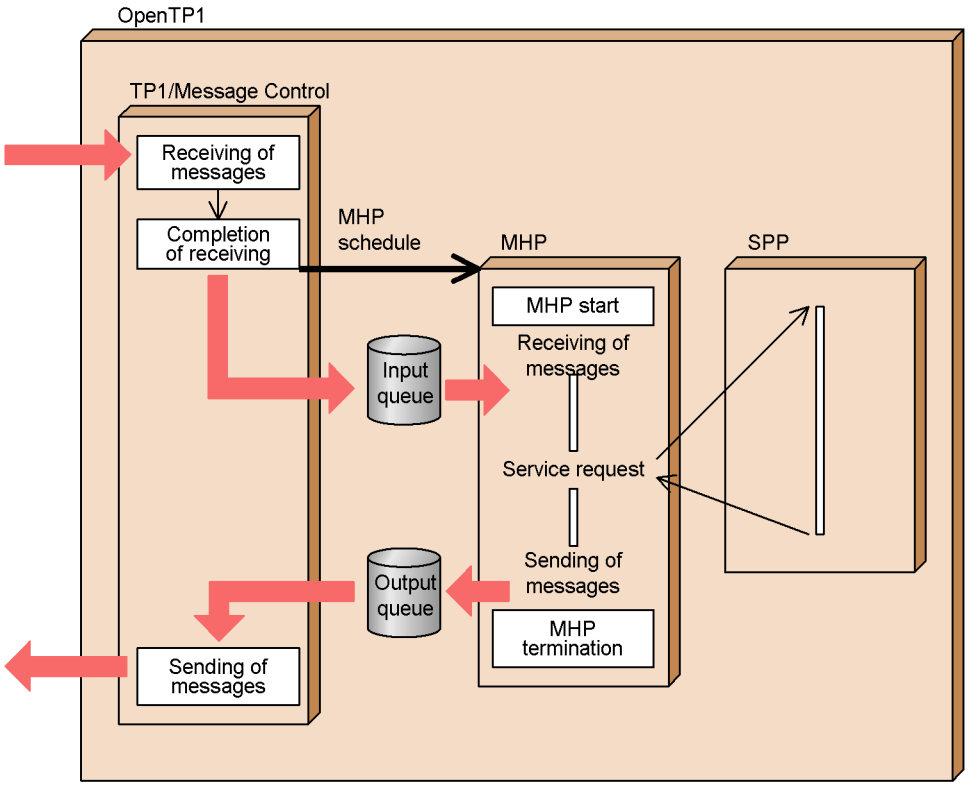
A UAP used for message exchange processing is called a *message handling program* (*MHP*). The MHP enables an own system connected with the MCF to communicate in message exchange mode. For details on message exchange processing see *3.6 Message exchange processing*.

The MHP can use the functions of OpenTP1 message exchange facilities. The MHP can also request SPP services from MHP processing by using the RPC.

To use the MHP, the MCF must be at the node at which the MHP exists.

The figure below shows the MHP.

Figure 1-13: Outline of MHP



(1) MHP configuration

Like the SPP, the MHP comprises a main function and service functions.

An application which is scheduled according to the application name in a message received by the MCF is created as a service function (a service program when COBOL language is used). Create more than one service function, link the service functions with a main function (a main program when COBOL language is used), and group all the functions into an executable file. Then, define the MHP executable file, comprising one main function and multiple service functions, as a service group in OpenTP1.

Since the facility for dynamic loading of service functions groups multiple services into a UAP shared library# before using them, it is not necessary to group them in the main function.

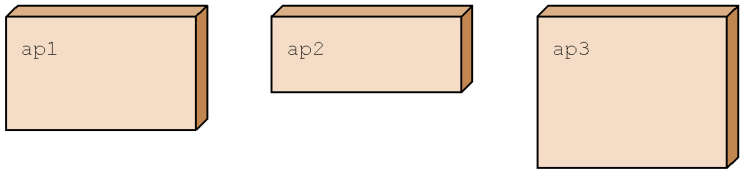
#

UAP shared library creation refers to linking the UAP object files created by compiling UAP source files and grouping them in a shared library.

The figures below show MHP configurations, the first of which uses a stub and the second of which uses dynamic loading of service functions.

Figure 1-14: MHP configuration (using a stub)

1. A service to be provided to the client UAP is created as a service function.



2. When a main function is created and the main function and the service function are compiled and linked, an MHP executable file is created.

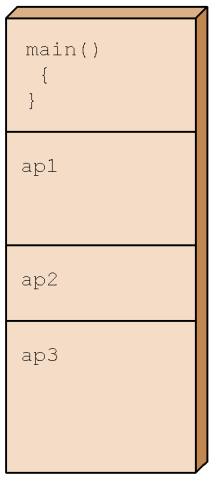
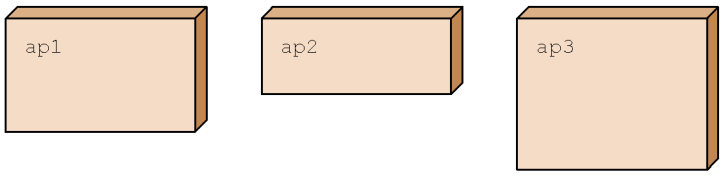
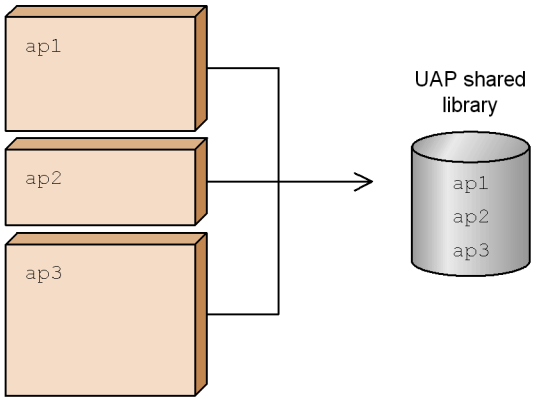


Figure 1-15: MHP configuration (using dynamic loading of service functions)

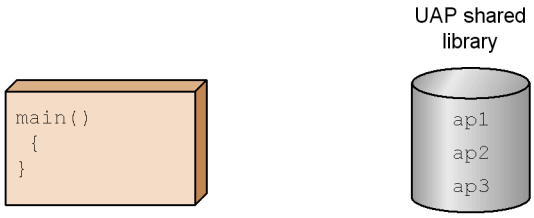
1. A service to be provided to the client UAP is created as a service function.



2. Service functions are grouped into a UAP shared library.



3. A main function is created and compiled to create an MHP executable file.
When the MHP is started, the service function is acquired from the UAP shared library created in Step 2 and is executed.



(2) MHP start

The MHP can be started either at the same time as OpenTP1 or anytime after OpenTP1 is started. If the first method is selected, MHP processing will start as soon as OpenTP1 starts. The starting time can be selected according to the purpose of the MHP.

(a) Starting at the same time as OpenTP1

Before starting OpenTP1, specify that the MHP is to start at the same time as OpenTP1. Specify the user server name of the MHP in the definition command `dcsvstart` for the user service configuration definition.

(b) Starting anytime after OpenTP1 is started

To start the MHP anytime after OpenTP1 is started, specify the user server name of the MHP as the argument to the `dcsvstart` command.

The MHP is initiated from the main function. It becomes ready to receive messages when the MHP service starting function (`dc_mcf_mainloop()`[`CBLDCMCF('MAINLOOP')`]) is normally executed. If the MHP terminates abnormally before the service starting function is called, processing is determined by the values assigned to the `hold` operand and `term_watch_time` operand in the relevant user service definition (or user service default definition).

(3) During MHP operation

The started MHP works as a previously specified process so that memory can be used efficiently. The started MHP may be activated as a resident process or nonresident process. In the former case, the MHP starts processing upon receiving a service request. Even in the latter case, a service request activates the process, thereby starting MHP processing.

For details about setting up UAP processes, see *1.3.5 Application program environment setup*.

(a) Starting a message handling MHP

After the MCF receives a message, the corresponding MHP process is started. The MHP is scheduled according to the application name in the first segment of the message. In the MCF application definition, correspond the application name to the service name for recognizing the UAP service in OpenTP1.

The message handling MHP can be started at either of the following times by using the function `dc_mcf_execap()`[`CBLDCMCF('EXECAP')`] by another UAP (MHP or SPP):

- When the UAP that called the function `dc_mcf_execap()` terminates normally (transaction commitment)
- When the specified number of seconds have passed after the UAP called the function `dc_mcf_execap()` or when the specified time comes

(b) Starting an MHP for an MCF event

If an MCF error or an MHP error occurs, a message is output for posting the error status from the MCF. This feature is called an *MCF event*. Create an MHP for an MCF event in case an MCF event is reported so that the MHP for an MCF event will perform

error recovery processing specific to the MCF event. Create an MHP for an MCF event in correspondence to the event code of an MCF event which might report. If the MCF event reports, the corresponding MHP for an MCF event is started. For details on MCF events see *3.10 MCF events*.

(4) MHP termination

The MHP is normally terminated when:

- OpenTP1 is normally terminated.
- The `dcsvstop` command in which the user server name of the MHP is specified is executed during OpenTP1 operation.

When one of the above events occurs, the function `dc_mcf_mainloop()` returns, thereby terminating the MHP.

Do not use the `kill` command to terminate the MHP process.

(5) Outline of MHP processing

Perform the following pre-processing for MHP main functions:

- Starts an application program.

  (`dc_rpc_open()` [`CBLDCRPC('OPEN  ')`])
- Opens an MCF environment.

  (`dc_mcf_open()` [`CBLDCMCF('OPEN  ')`])
- Starts an MHP service.

  (`dc_mcf_mainloop()` [`CBLDCMCF('MAINLOOP')`])
- Closes an MCF environment.

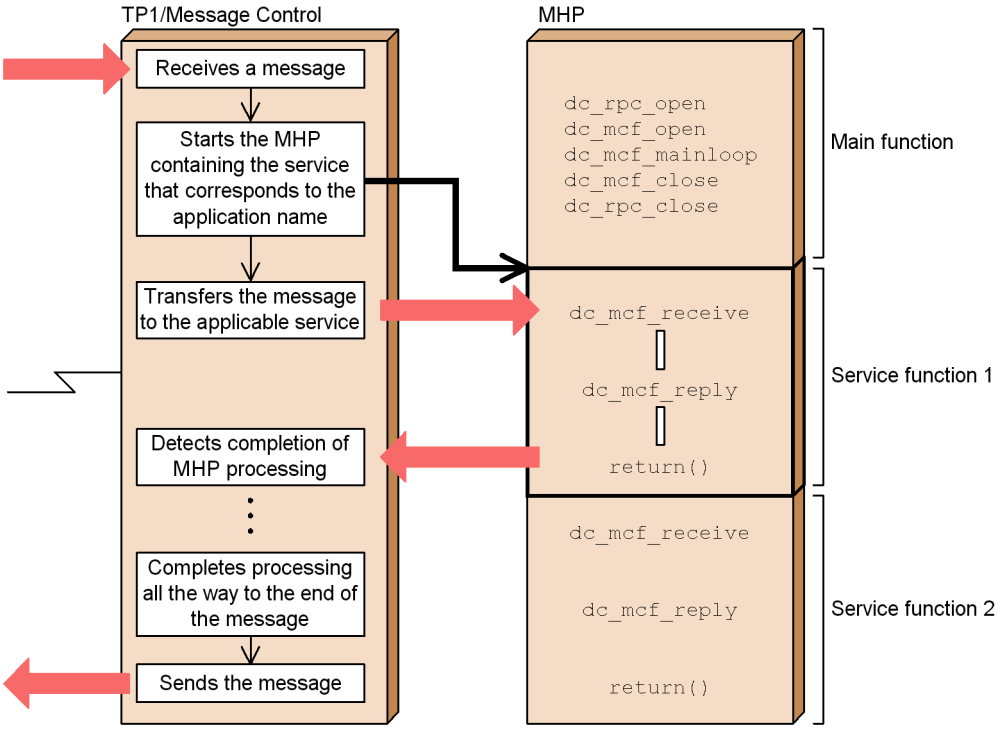
  (`dc_mcf_close()` [`CBLDCMCF('CLOSE  ')`])

The figure below shows MHP processing.

(6) Note

You cannot call the MHP service functions using the RPC.

Figure 1-16: Outline of MHP processing (C language)



1.2.4 UAP that handles offline work

A UAP for handling batch jobs can be created if necessary. A UAP which handles the initialization, allocation, and deletion of DAM files as well as batch jobs is executed under the offline environment.

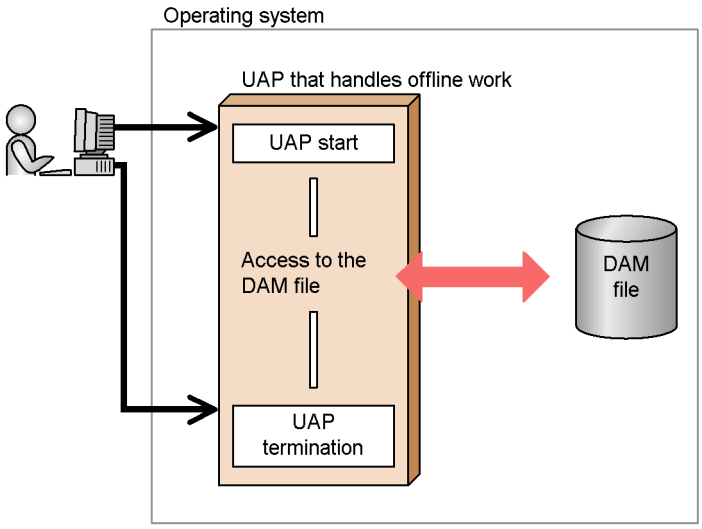
OpenTP1 facilities which can be used by a UAP that handles offline work are as follows:

- Facility to process physical `DAM` files
- Facility to edit journal data in the output file of the `jnlrput` command.

UAP that handles offline work cannot use the OpenTP1 functions which are used in online mode. The RPC for service requests cannot be used between a UAP that handles offline work and a UAP (SUP, SPP, or MHP) operating under the online environment. Services to be provided for another UAP cannot be created by a UAP that handles offline work.

The figure below shows a UAP that handles offline work.

Figure 1-17: Outline of UAP that handles offline work



(1) Starting and terminating UAP that handles offline work

Use the shell to start a UAP that handles offline work. Users are to manage the start and termination of UAP that handles offline work.

1.3 Creation of application programs

This section explains the procedure for creating OpenTP1 UAPs. The figure below shows the procedure from UAP creation to execution.

Figure 1-18: Procedure for UAP creation (when using a stub)

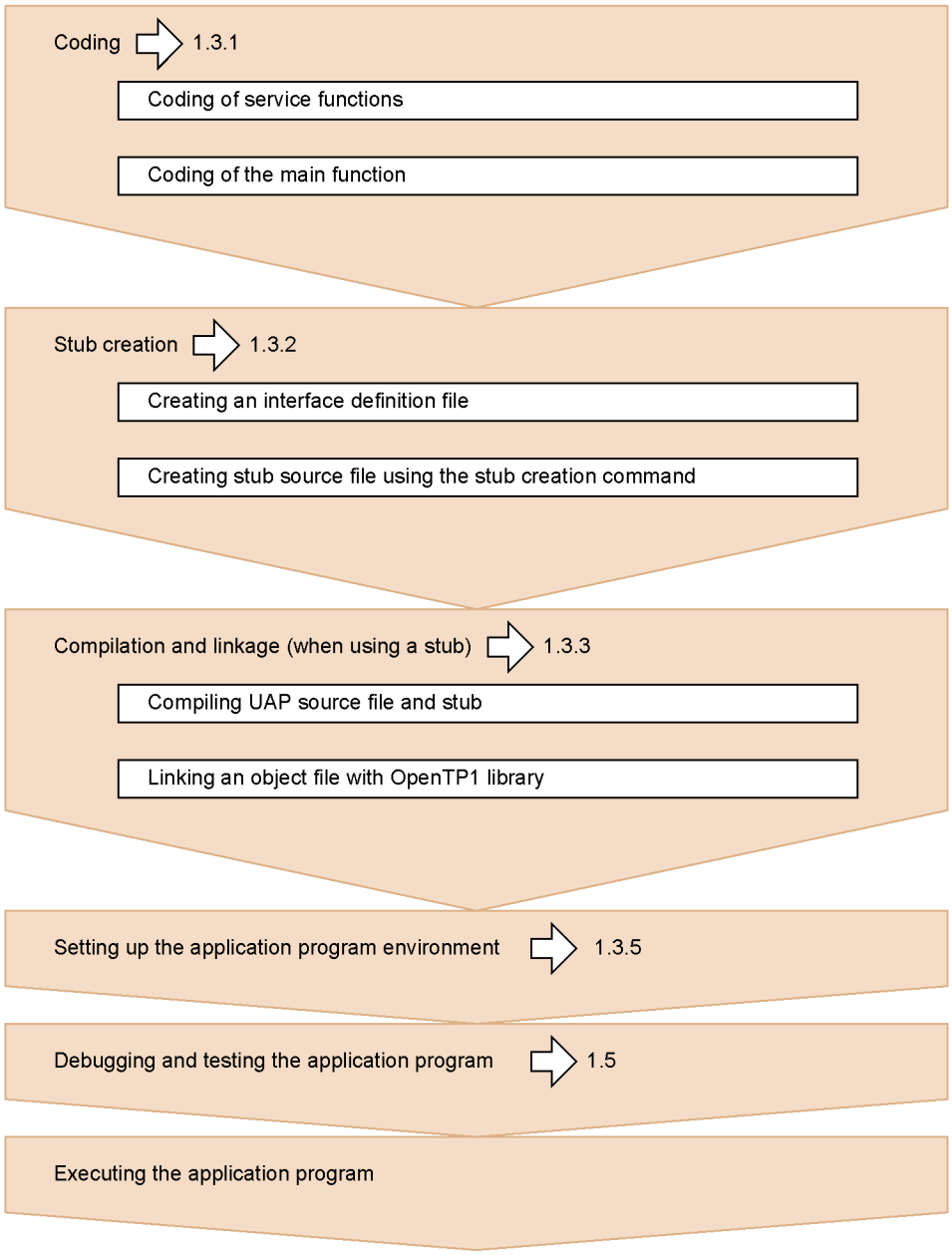
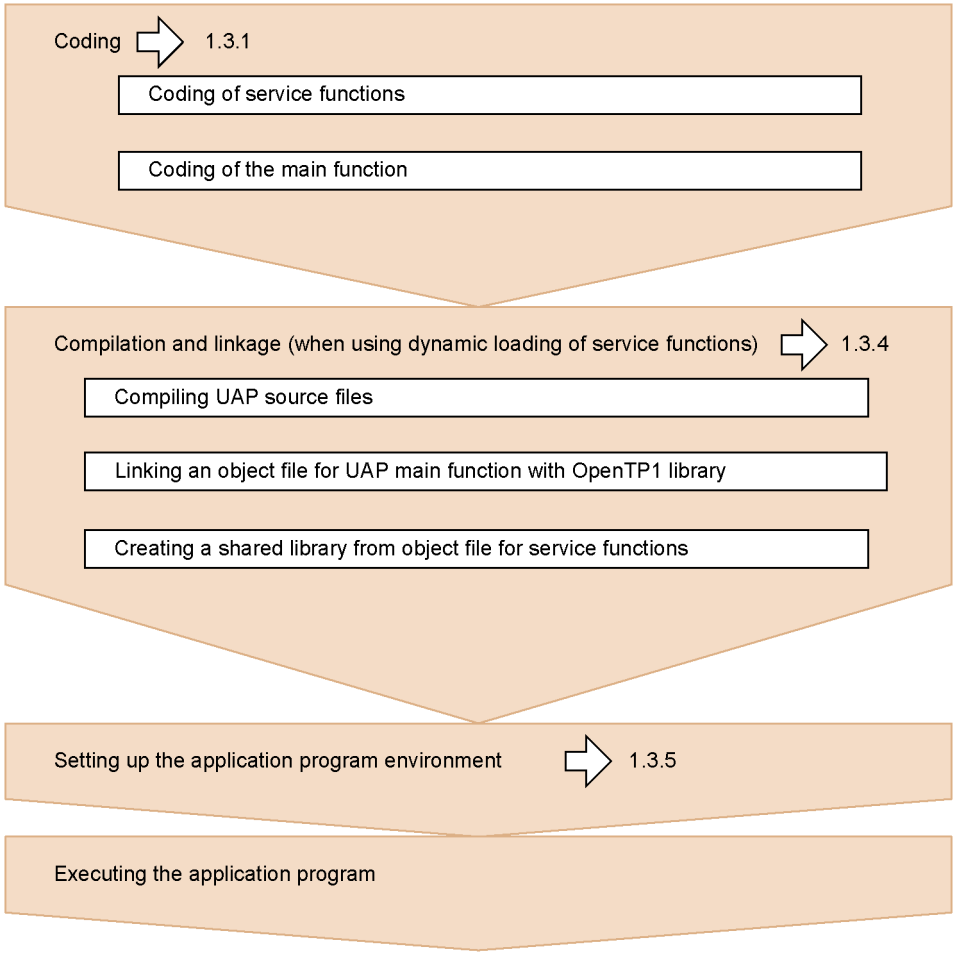


Figure 1-19: Procedure for UAP creation (when using dynamic loading of service functions)



1.3.1 Coding

Use the C, C++ or COBOL language when coding OpenTP1 UAPs. Not only OpenTP1 facilities, but also standard OS facilities and structured query language (SQL) can be used for OpenTP1 UAPs. For details of the coding rules, see the applicable *OpenTP1 Programming Reference* manual. For details on the SQL coding rules, see the appropriate reference manual.

(1) Coding in C or C++

(a) When using C

Code the UAP in either the ANSI C format or the pre-ANSI K&R format (Classic C).

To use an OpenTP1 facility from the UAP, call the corresponding OpenTP1 library function.

(b) When using C++

Code the UAP in the ANSI C format according to the C++ specification. To use an OpenTP1 facility from the UAP, call the corresponding OpenTP1 library function. Note that linking the UAP coded in C++ causes the OpenTP1 library function to be linked and operated as a C function because the header file (dc*xxx*.h) specifies that OpenTP1 library functions should be linked to elements written in C.

(c) How to use OpenTP1 functions

As in the case of OS-provided standard functions, when calling functions, set their arguments.

Whether a function has been normally executed can be determined from the return value from the function. Some functions give return values, whereas others do not.

The figure below shows UAP coding in C.

Figure 1-20: Outline of UAP coding in C

```
#include <stdio.h>
#include <string.h>
#include <dcrpc.h>
#include <dctrn.h>

main()
{
/* UAP start */

    rc = dc_rpc_open(DCNOFLAGS);
    if(rc != DC_OK){
        printf("dc_rpc_open failed!!");
        goto PROG_END;
    }

/* Remote procedure call  */

    rc = dc_rpc_call("svr01","svr01",…);
    if(rc != DC_OK){
        printf("Service request failed!!")
        goto PROG_END;
    }

/* UAP termination */
    PROG_END:
    dc_rpc_close(DCNOFLAGS);
    printf("Processing completed. \n");
    exit(0);
}
```

Loads the header file to be used in program processing. OpenTP1 header files begin with `dc`.

A main function begins with `main()`; a service function begins with a function name.

A branch condition can be added to the library function to return a return value.

A library function that has no return value is called as is.

(2) Coding in COBOL

COBOL/2# or COBOL85 is available for UAP coding in the COBOL language. To use OpenTP1 facilities from the UAP, use COBOL-UAP creation programs corresponding to OpenTP1 library functions. The COBOL-UAP creation program is called by the CALL statement in COBOL and transfers control from UAP processing to the OpenTP1 library.

The results of CALL statement execution can be identified by the numeric value returned (status code). Some COBOL-UAP creation programs do not return status codes.

The figure below shows UAP coding in COBOL.

Figure 1-21: Outline of UAP coding in COBOL

```
*
 IDENTIFICATION DIVISION.
*
 PROGRAM-ID. MAIN.
*
 DATA DIVISION.
 WORKING-STORAGE SECTION.
 01  ARG1.
     02 REQUEST      PIC X(8) VALUE SPACE.
     02 STATUS-CODE  PIC X(5) VALUE SPACE.

             :
*
 PROCEDURE DIVISION.
*
 MOVE 'CALL'  TO REQUEST OF ARG2.
 MOVE 'SPP01' TO G-NAME  OF ARG2.
 MOVE 'SVR01' TO S-NAME  OF ARG2.
             :
  CALL 'CBLDCRPC' USING ARG2 ARG3 ARG4.
     IF STATUS-CODE OF ARG2 NOT = '00000' THEN
             :
     END-IF.
             :
```

DATA DIVISION# defines the areas to be used by the OpenTP1 COBOL-UAP creation program.

The necessary values are specified in the defined areas.

The COBOL-UAP creation program is called by the CALL statement.

#

In Base samples, the *DATA DIVISION template* can be used for each COBOL-UAP creation program. For details on the DATA DIVISION template, see *8.8 COBOL language templates*.

(3) Note

- Do not use coding that creates multiple threads, because it might cause problems during UAP termination.
- OpenTP1-provided APIs are not thread-safe. They are internally controlled by their own threads. Therefore, all OpenTP1 APIs must be run on the main thread. No OpenTP1 APIs can be issued outside the main thread except for the following APIs:
  - TP1/EE C language interfaces that have names beginning with ee_
  - TP1/EE COBOL language interfaces that have names beginning with CBLEE

1.3.2 Creating stubs

UAPs used with OpenTP1 require libraries for fulfilling inter-UAP service requests.

One of these libraries is called a *stub*. Information about UAP service is specified in the stub. Information about the destination of communication is created in some cases.

For details on how to create stubs, see the applicable *OpenTP1 Programming Reference* manual.

(1) Types of stub linked to application programs

Stubs are linked to the server or client UAP.

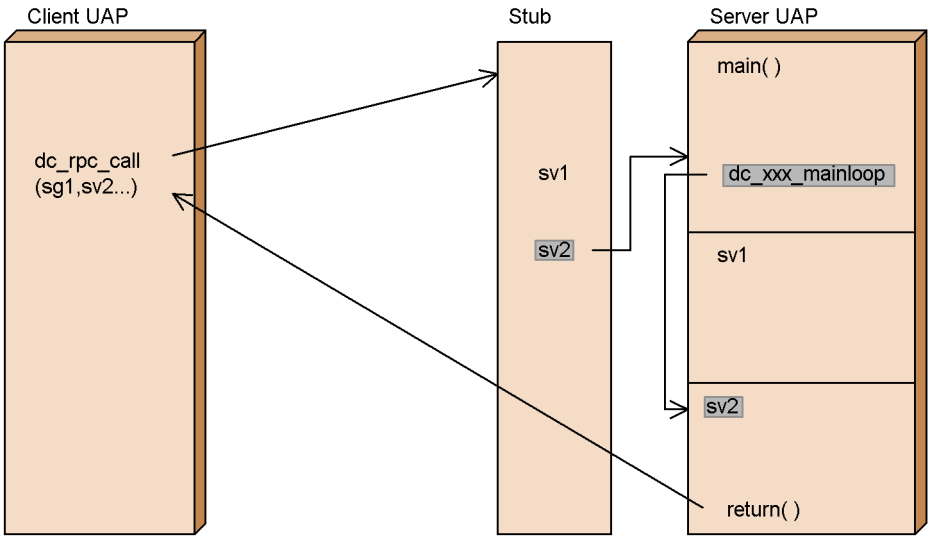
(a) Stub linked to server UAP

A stub linked to the server UAP, working in cooperation with service distribution functions, makes the UAP ready to offer its service. The service distribution functions, which are called by the main function of the server UAP, are listed below.

- For SPPs: `dc_rpc_mainloop()` [`CBLDCRSV('MAINLOOP')`]
- For MHPs: `dc_mcf_mainloop()` [`CBLDCMCF('MAINLOOP')`]

The figure below shows a stub linked to the server UAP.

Figure 1-22: Stub linked to server UAP



(b) Stub linked to client UAP

A stub linked to the client UAP enables communication with a server UAP according to information specified about that server UAP. The client UAP requires a stub only when the XATMI interface is used for communication. When OpenTP1 RPCs are in use, the client UAP requires no stub.

(2) UAPs requiring a stub

Whether a UAP requires a stub depends on the type of UAP and the communication method used.

- UAPs using OpenTP1 remote procedure call (SUP, SPP)

  SPPs require a stub. SUPs require no stub.

- MHP

  MHPs require a stub. Create a stub with a procedure similar to that in the case of SPPs.

- Client/server mode communication through XATMI interface

  Both client UAPs (SUP, SPP) and the server UAP (SPP) require stubs.

No stub is required for UAP that handles offline work because they do not contain a service function.

(3) Stub creation procedure

Before creating a stub, create a file (RPC interface definition file[#]) containing information about the UAP definition. Execute the stub creation command with its argument specified to identify this file. The following stub creation commands are available:

- When the UAP is for TCP/IP communication: `stbmake` command
- When the UAP is for OSI TP communication: `tpstbmk` command

When the stub creation command is executed, a stub source file (source file in C) is created. Compile this file with the C compiler and link it to the object file of the UAP.

When an MHP is coded in ANSI C or C++ style, define `DCMHP` at compilation of the stub which is linked to the MHP.

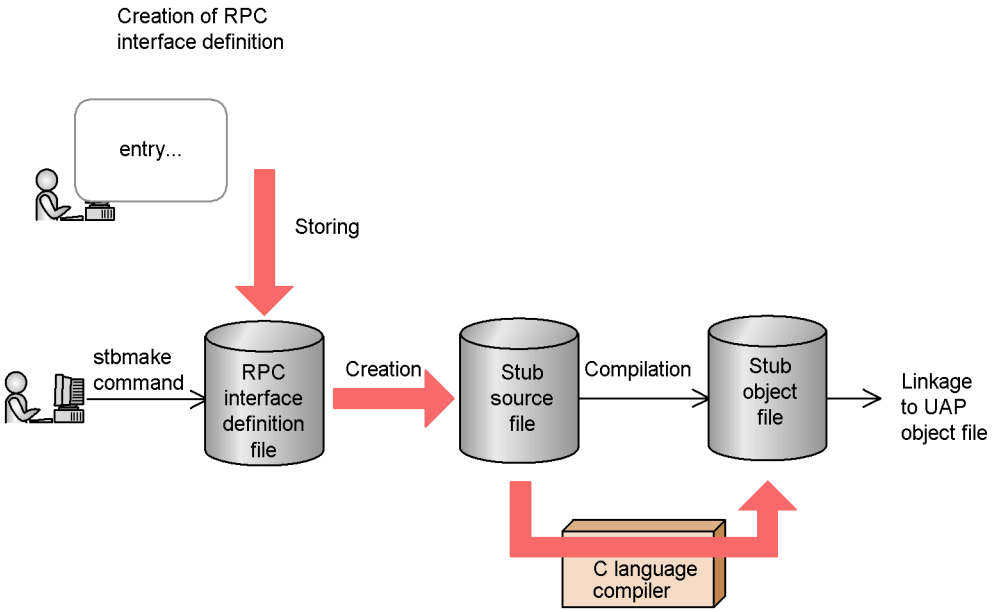
When modifying the stub, create the UAP from scratch. Modify the RPC interface definition file, recreate the stub, and link it to the object file of the recompiled UAP.

#

For stubs compliant with the XATMI interface, this file is referred to as the *XATMI interface definition file*.

The figure below shows the stub creation procedure.

Figure 1-23: Stub creation procedure



1.3.3 Compilation and linkage (when using a stub)

This subsection explains the procedure for compiling and linking created programs. The UAP is compiled and linked into an executable file. For details on the compilation and linkage procedure, see the applicable *OpenTP1 Programming Reference* manual.

(1) Compilation

The following programs must be compiled:

- UAP source file (main and service functions)
- Stub (if required for the UAP)

Use the C language compiler to compile source programs written in C and the COBOL language compiler to compile source programs written in COBOL.

(2) Linkage

Compiled object files are linked to the OpenTP1 library and other necessary files. If a non-OpenTP1 resource manager is used, it must be linked to the library specified by the non-OpenTP1 resource manager. To use a non-OpenTP1 resource manager with the XA interface, link the library to the UAP by performing the following steps:

1. Specify the resource manager identifier for the non-OpenTP1 resource manager in the `trnmkobj` command and execute this command to create a transaction

control object file.

2. Link the object file to the UAP.

(3) Note

If the OS is HP-UX, the bind mode for linkage must be specified as `immediate`. If an executable file created in another mode is used as an OpenTP1 UAP, the system operation is unpredictable. To check that the bind mode of the created UAP is `immediate`, use the `chatr` command of the OS.

1.3.4 Compilation and linkage (when using dynamic loading of service functions)

This subsection explains the procedure for compiling and linking created programs, and how to incorporate them into a UAP shared library#.

First, the main function of the UAP is compiled and linked into an executable file. Next, the UAP's service functions are incorporated into a UAP shared library#. For details on the compilation and linkage procedure, see the applicable *OpenTP1 Programming Reference* manual.

#

Refers to the concept of compiling UAP source files to produce UAP object files, which are then linked to create a shared library.

(1) Compilation

The following programs must be compiled:

- UAP source file (main and service functions)

Use the C language compiler to compile source programs written in C, and the COBOL language compiler to compile source programs written in COBOL

(2) Linkage

The object file created by compiling the source files for the UAP main function is linked to the OpenTP1 library and other necessary files. If a non-OpenTP1 resource manager is used, the object file must be linked to the library specified by the non-OpenTP1 resource manager. To use a non-OpenTP1 resource manager with the XA interface, link the library to the UAP by performing the following steps:

1. Specify the resource manager identifier for the non-OpenTP1 resource manager in the `trnmkobj` command and execute this command to create a transaction control object file.
2. Link the object file to the UAP.

(3) Creating a UAP shared library

Incorporate the object file you created by compiling the source file for the service functions into the shared library. In the same manner as in (2) above, the source file is linked to the OpenTP1 library and other necessary files. See the TP1/Server Base sample file (`make_svdl`) for compilation and linkage options.

(4) Note

If the OS is HP-UX, `immediate` must be specified for the bind mode for linkage. If an executable file created in another bind mode is used as an OpenTP1 UAP, system operation cannot be guaranteed. To check whether the bind mode of a created UAP is `immediate`, use the `chatr` command of the OS.

1.3.5 Application program environment setup

An appropriate environment must be set up before the executable file of the created UAP can be used as an OpenTP1 user server.

(1) Directory containing UAP

The executable file of the created UAP is stored in the `$DCDIR/aplib/` directory (where `$DCDIR` represents the OpenTP1 home directory).

(2) UAP registration

The executable file of the UAP is registered with OpenTP1. An OpenTP1 UAP is called a *user server* because it offers services.

(a) UAP registration method

When registering a UAP with OpenTP1, set up the environment in which the UAP will run. The setting method is as follows:

- TP1/Server Base

  Set a UAP execution environment in the user service definition.

- TP1/LiNK

  Use the `dcsysset -u` command for UNIX or the **Application Control** icon for Windows.

(b) User server name

The UAP name (user server name) under OpenTP1 is as follows:

- TP1/Server Base

  File name of the user service definition

- TP1/LiNK

  When setting up a UAP execution environment, give any user server name

associated with the executable file name.

(3) UAP names

Names given to programs created as UAPs are explained below.

- UAP executable file name

  This name is set in an option to the linkage command when the UAP object file is linked to the library.

- User server name

  This name is set when the UAP is registered with OpenTP1. It is specified as the argument to the `dcsvstart` command. It is 1 to 8 characters long.

- Service group name and service name

  These names are specified as arguments to functions which are used to request service through OpenTP1 remote procedure calls or communication via the XATMI interface. When the user server name is registered with OpenTP1, these names are also specified.

  Each UAP executable file is given a service group name.

  Each service function is given a service name which works as a function name.

- Application name

  After a message is received through TP1/Message Control, it is processed by the application identified by this name. MHP service functions are registered under application names given to them. The relationship between service names and application names is specified in the MCF application definition.

1.3.6 User server load balancing and scheduling

This subsection explains the multiserver facility provided for efficient use of UAPs (user servers) and how to schedule UAPs.

When OpenTP1 system services or user servers are run, the OS work area is used. Operation performed on this work area is called a *process*. The process generated by running a user server is specifically called the *user server process*, the *UAP process*, or simply the *process*. OpenTP1 controls the total number of processes in use so that the number of processes will not increase or decrease beyond appropriate levels.

Before the user server process can be controlled, the user server must be started. The user server must be started at the same time as OpenTP1 or by executing the `dcsvstart -u` command.

(1) Multiserver facility

When a user server running to handle a service request receives another service request, user server processing for the new service request can be performed by a new

process. In this way, one user server can run another process in parallel to the current process. This is referred to as the *multiserver facility*.

The multiserver facility is available to SPPs that use the schedule queue (queue-receiving servers), not to servers that receive requests from socket. For a server that receives requests from socket, specify only one process to be used.

(2) Resident and nonresident processes

UAP processes for which multiserver facility is specified can be acquired either always during OpenTP1 operation or dynamically. An always acquired process is called a *resident process*. A process which is not acquired during OpenTP1 operation, but is started when necessary is called a *nonresident process*.

If processes are specified as nonresident, the memory area in the OpenTP1 system can be used efficiently. When a process is specified as resident, its user server processing is quicker than when it is specified as nonresident.

If free memory space is unavailable in the system, a nonresident process will start after the currently running nonresident process terminates.

(3) Method for process setup

The number of processes to be started by the user server as resident/nonresident processes is set up in advance. The specified number of processes can be started in parallel. The setup method is as follows:

- TP1/Server Base

  Specify the total number of processes to be used (the number of resident processes and the number of nonresident processes) for `parallel_count` in the user service definition.

- TP1/LiNK

  When setting up a UAP execution environment, specify the number of processes to be used (the number of resident processes and the number of nonresident processes).

If more than one resident process is specified, the specified number of processes will be started in parallel. If more than one nonresident process is specified, the specified number of processes can be started dynamically.

(4) Multiserver load balance

The number of nonresident processes can be increased or decreased according to the number of service requests in the schedule queue. This is called the *multiserver load balancing facility*.

When to start a nonresident process is determined by the value assigned to the `balance_count` operand in the user service definition. When the number of service requests in the schedule queue exceeds the product of the value assigned to the

balance_count operand and the number of active processes, the OpenTP1 starts a nonresident process. When the number of service requests in the schedule queue drops below the product of the value assigned to the balance_count operand and the number of active processes, the OpenTP1 terminates a nonresident process.

The method for specifying the value determining when to start a nonresident process is as follows:

- TP1/Server Base

  Assign a value to balance_count in the user service definition.

- TP1/LiNK

  The value assigned to the balance_count operand is equal to the maximum number of remaining request services which was specified when the UAP execution environment was set up.

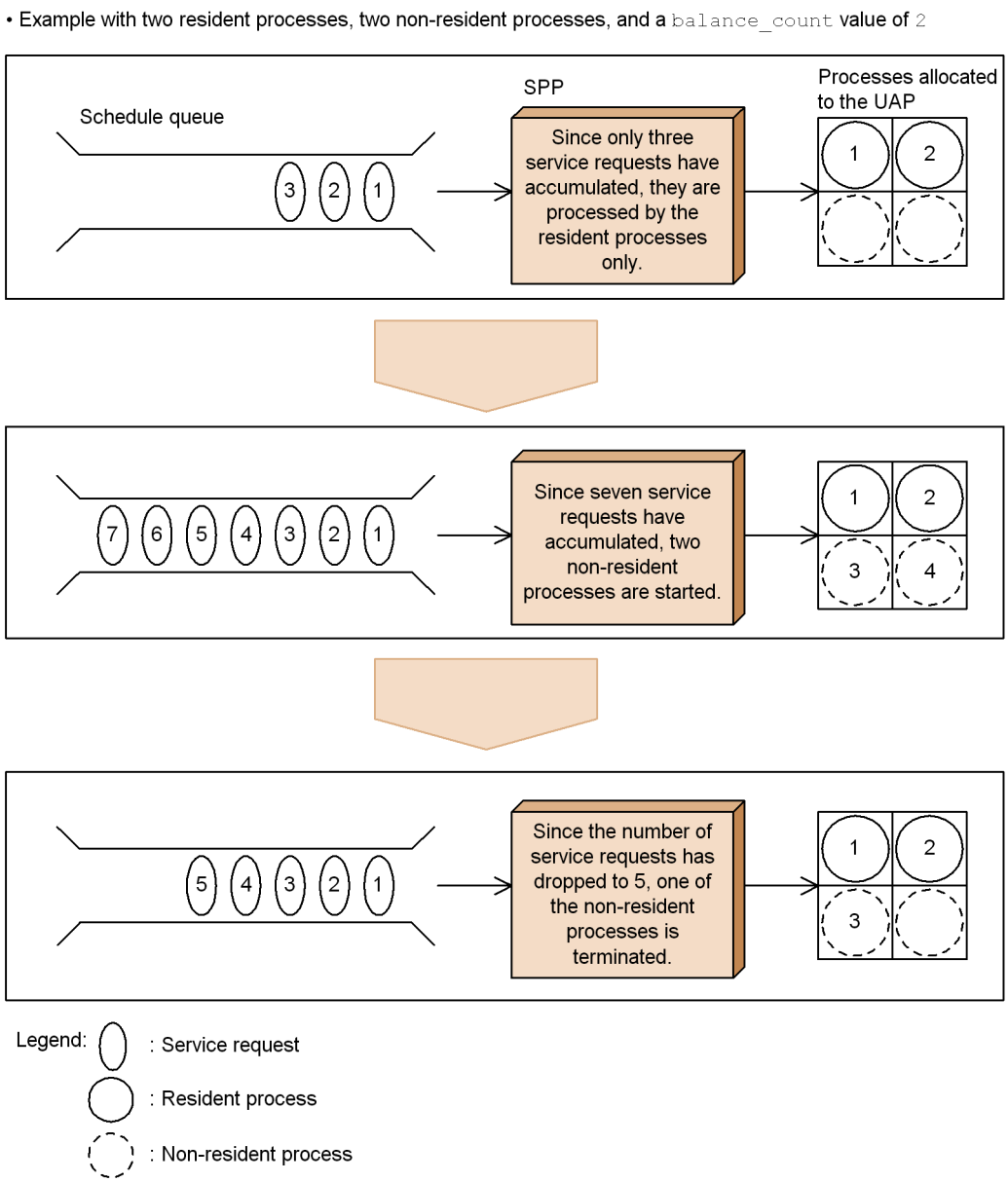
(5) Schedule priority

Each user server can be given a schedule priority. Nonresident processes of a user server given a higher schedule priority will be scheduled with priority over other nonresident processes.

When processes to be used with a user server are set up, their schedule priorities are also set up.

The figure below shows a process load balancing.

Figure 1-24: Process load balancing



(6) Internode load-balancing facility

When user servers having the same service group name are placed on multiple nodes, a service request can be handled by any user server on any node. As the result, the load

can be distributed among nodes. This facility is called *internode load-balancing facility*. Particular environment setup is not required to use this facility. OpenTP1 distributes the load automatically if only the user servers having the same service group name on these nodes are activated.

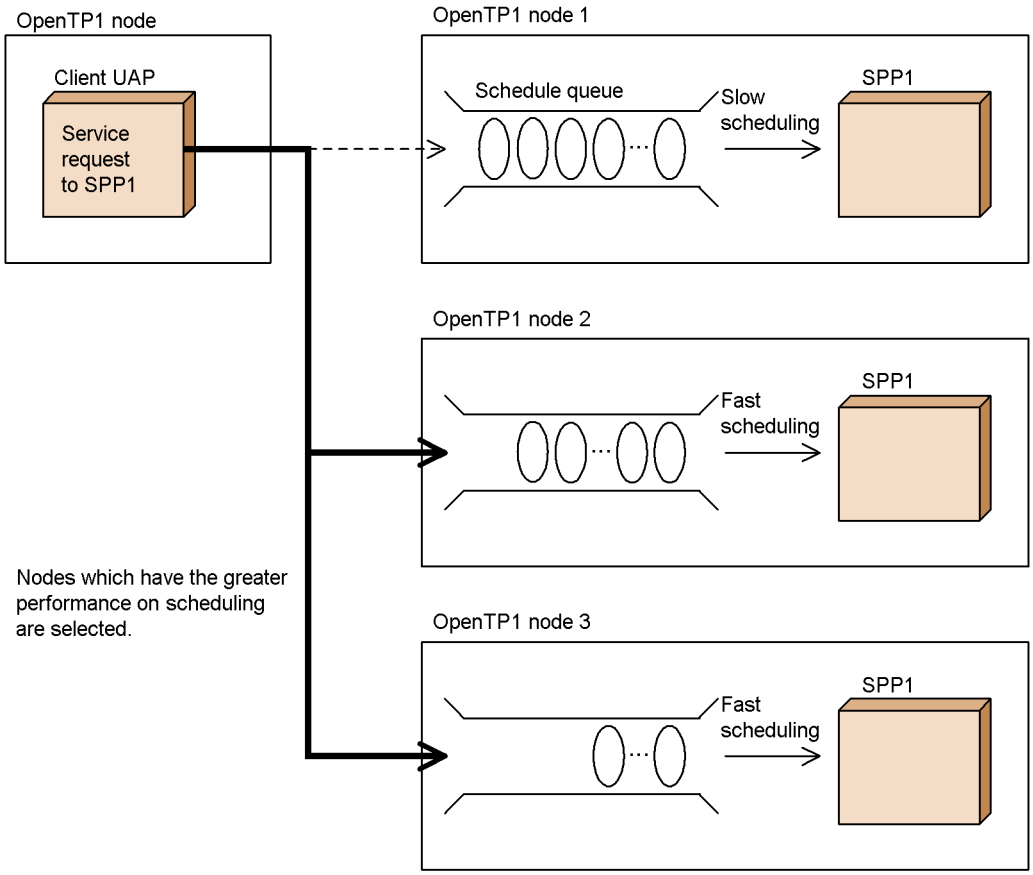
Consider a service group in the OpenTP1 system which contains some user servers that use the multi-scheduler facility, and some that do not. In this case, even when there is a significant load on the user servers that use the multi-scheduler facility, the load is not distributed to the user servers that do not use the multi-scheduler facility. To distribute the load to user servers that do not use the multi-scheduler facility, specify the `-t` option in the `scdmulti` definition command of the schedule service definition. For details on the `scdmulti` definition command, see the description of the schedule service definition in the manual *OpenTP1 System Definition*.

The internode load-balancing facility can distribute loads to 128 or less nodes.

The internode load-balancing facility distributes the load to the node which can process the request more efficiently according to the schedule status of the nodes. When the user server on the node which contains the UAP requesting the service is to be scheduled with priority, specify `Y` in the `scd_this_node_first` operand of the schedule service definition for the node.

The figure below shows the outline of the internode load-balancing facility.

Figure 1-25: Outline of internode load-balancing facility



(7) Extended internode load-balancing facility

You can define the following specifications:

- Schedule rate for LEVEL0 nodes

  You can define the schedule rate for nodes whose load level is LEVEL0 by specifying an appropriate value in the `schedule_rate` operand of the schedule service definition.

- Load monitoring interval time

  You can define the load monitoring interval time for each service group by specifying an appropriate value in the `loadcheck_interval` operand of the user service definition or the user service default definition.

- Thresholds for load levels

You can define the thresholds for each service group by specifying appropriate values in the `levelup_queue_count` and `leveldown_queue_count` operands of the user service definition or the user service default definition. These thresholds will determine load levels based on the number of service requests remaining.

- Number of retries on a communication error

  If a communication error occurs during service requests scheduling, the process usually returns with an error and does not attempt re-scheduling.

You can define the number of retries attempted in order to schedule requests to nodes other than where a communication error occurred by specifying an appropriate value in the `scd_retry_of_comm_error` operand of the schedule service definition.

TP1/Extension 1 must be installed before you can use this facility. Note that operation will be unpredictable if you run the facility while TP1/Extension 1 is not installed.

(8) Multi-scheduler facility

When a client UAP requests a service provided by a queue-receiving server (SPP that uses the schedule queue) on a remote node, the scheduler daemon on the node containing the request destination server receives the service request message and stores it in the schedule queue of the relevant queue-receiving-server. A scheduler daemon is a system daemon that provides a schedule service.

The scheduler daemon is a single process provided on each OpenTP1 system. Therefore, as systems become larger and machines and networks boast increasingly better performances, the conventional scheduler daemon may experience difficulty scheduling messages efficiently. If the conventional scheduler daemon cannot schedule messages efficiently, see *C. Examples of System Configurations Requiring Consideration of the Multi-Scheduler Facility*.

OpenTP1 provides a daemon exclusively for receiving service requests (referred to below as the *multi-scheduler daemon*) in addition to the conventional scheduler daemon (referred to below as the *master scheduler daemon*). The multi-scheduler daemon prevents scheduling delays caused by contention during receive processing. It does this by starting multiple processes and running receive processing for service request messages in parallel. This facility is called the *multi-scheduler facility*.

TP1/Extension 1 must be installed before you can use this facility. Note that operation will be unpredictable if you run the facility while TP1/Extension 1 is not installed.

To use the multi-scheduler facility, you must specify the following definitions:

On the RPC receiver:

Schedule service definition `scdmulti`

User service definition `scdmulti`

On the RPC sender:

User service definition `multi_schedule`

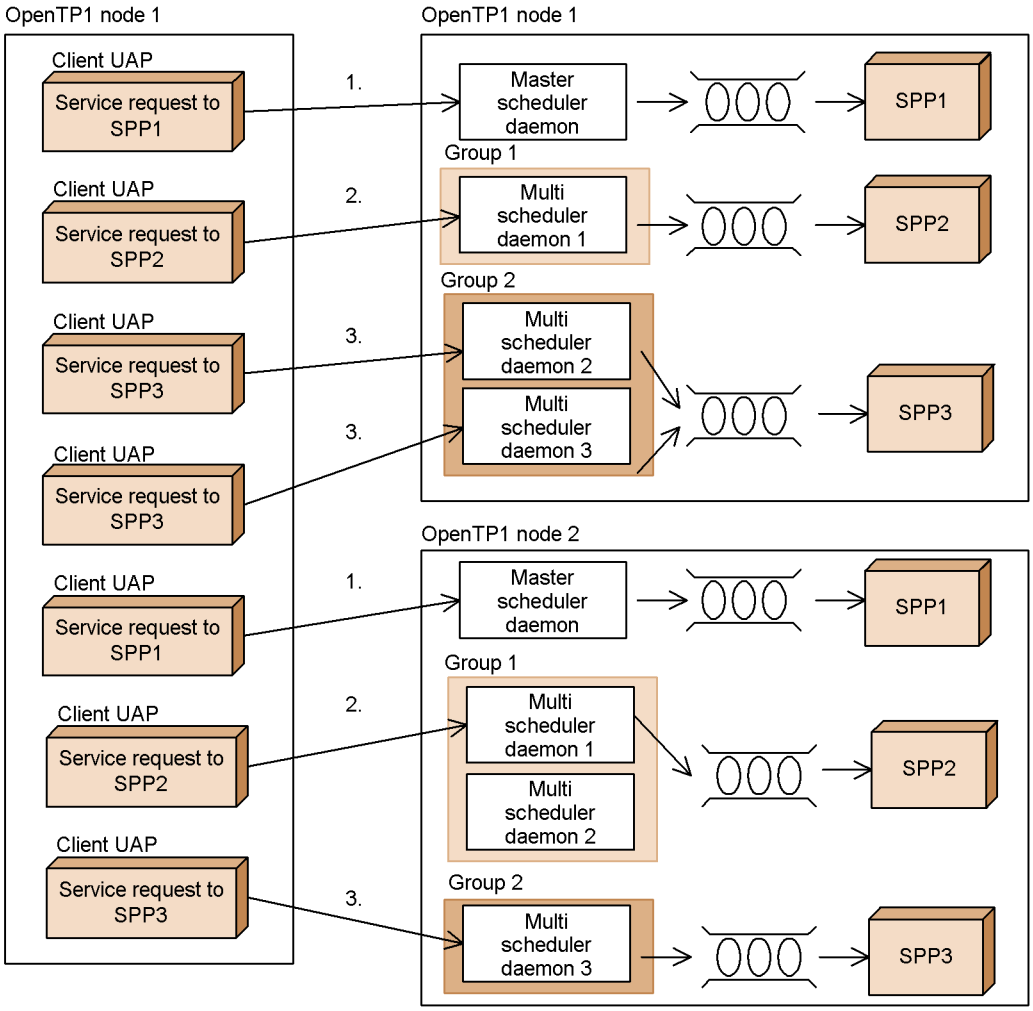
You can also group several multi-scheduler daemons for each queue-receiving server. This prevents contention when different servers receive service request messages. To group and use multi-scheduler daemons, you must specify the `-g` option in the user service definition `scdmulti` on the server.

When OpenTP1 starts, it starts the multi-scheduler daemon specified in the definition at the well known port number in addition to the master scheduler daemon. It starts the multi-scheduler daemon as a system daemon providing schedule services. For details on requesting a service using the multi-scheduler facility provided by TP1/Client, see the manual *OpenTP1 TP1/Client User's Guide TP1/Client/W, TP1/Client/P*.

For details on RPC that uses the multi-scheduler facility, see *2.1.16 RPC with the multi-scheduler facility*.

The figure below shows an example of using the multi-scheduler facility.

Figure 1-26: Example of using the multi-scheduler facility



- Because SPP1 handles short service request messages, set it to use the master scheduler daemon instead of the multi-scheduler facility (1. in the figure).
- Because SPP2 handles long service request messages, it is distributed between nodes 1 and 2 of OpenTP1. Set it to use multi-scheduler daemon 1 of Group 1 located on node 1, or multi-scheduler daemons 1 and 2 of Group 1 located on node 2 (2. in the figure).
- SPP3 handles short service request messages, but the number of service requests is large. Therefore, SPP3 is distributed between nodes 1 and 2 of OpenTP1. Set it to use multi-scheduler daemons 2 and 3 of Group 2 located on node 1, or multi-scheduler daemon 3 of Group 2 located on node 2 (3. in the figure).

1.4 OpenTP1 library functions

1.4.1 Application programming interface facilities

The following facilities are available through OpenTP1 library functions:

(1) Basic OpenTP1 facility (TP1/Server Base, TP1/LiNK)

- Remote procedure call

  A method similar to function calls can be used for communication between UAPs.

- Transaction control

  UAP processes can be controlled as transactions.

- System operation management

  UAPs can execute commands and report the status of user servers.

- Message log output

  UAPs can output any user information as message logs.

- User journal acquisition

  User journals (UJs) can be acquired in system journal files.

- Journal data editing

  Journal data in the file containing the execution result of the `jnlrput` command can be edited.

- Real-time statistical information acquisition

  Real-time statistical information can be acquired in an arbitrary section in the UAP.

(2) Facilities available with TP1/Message Control

- Message exchanging

  TP1/Message Control allows message exchange mode communication in a wide-area network and between systems interconnected through TCP/IP.

(3) Facilities available for files

- DAM file service

  Direct files can be used as OpenTP1-dedicated user files.

- TAM file service

  Table access based direct files can be used as OpenTP1-dedicated user files.

- ISAM file service#

  Indexed sequential files complying with the X/Open ISAM model can be used.

- IST service (TP1/Shared Table Access)

  One or more tables (internode shared tables) in shared memory can be shared between two or more OpenTP1 systems. For the IST service, the entity of each user file does not exist and data is stored in an internode shared table in memory instead.

- Lock for resources

  OpenTP1 APIs can lock any files (UNIX files).

#

For details on the ISAM file service, see the manual *Indexed Sequential Access Method ISAM*.

(4) X/Open-compliant application program interfaces

- XATMI interface

  This interface allows client/server mode communication between X/Open-compliant APIs.

- TX interface

  X/Open-compliant APIs can control transactions.

(5) Facilities that are used in special styles

Listed below are functions for facilities that are used in special styles.

(a) Facilities available with TP1/Multi

- Multinode facility

  Various facilities are available to UAPs in cluster/parallel OpenTP1 systems.

(b) Facilities available with online tester (TP1/Online Tester)

- Management of online tester

  The user server test status can be obtained by calling an appropriate function from the UAP.

1.4.2 List of OpenTP1 library functions

(1) List of library functions

Tables 1-1 to 1-5 list the OpenTP1 library functions.

Table 1-1: OpenTP1 library functions (basic OpenTP1 facilities)

Facility		Library function name	
		C language library	COBOL-UAP creation program
Remote procedure call	Start a UAP	dc_rpc_open	CBLDCRPC('OPEN ')
	Start an SPP service	dc_rpc_mainloop	CBLDCRSV('MAINLOOP')
	Request a remote service	dc_rpc_call	CBLDCRPC('CALL ')
	Invoke a remote service with a communication destination specified#1	dc_roc_call_to	--
	Receive processing result in asynchronous mode	dc_rpc_poll_any_replies	CBLDCRPC('POLLANYR')
	Acquire the descriptor of an asynchronous RPC request which has encountered an error	dc_rpc_get_error_descriptor	CBLDCRPC('GETERDES')
	Reject the receiving of processing results	dc_rpc_discard_further_replies	CBLDCRPC('DISCARDF')
	Reject reception of selected processing results	dc_rpc_discard_specific_reply	CBLDCRPC('DISCARDS')
	Retry a service program	dc_rpc_service_retry	CBLDCRPC('SVRETRY')
	Set a schedule priority of service request	dc_rpc_set_service_prio	CBLDCRPC('SETSVPRI')
	Reference the schedule priority of service request	dc_rpc_get_service_prio	CBLDCRPC('GETSVPRI')
	Reference the service response waiting interval	dc_rpc_get_watch_time	CBLDCRPC('GETWATCH')
	Update the service response waiting interval	dc_rpc_set_watch_time	CBLDCRPC('SETWATCH')
	Acquire the node address of a client UAP	dc_rpc_get_callers_address	CBLDCRPC('GETCLADR')
	Acquire the node address of a gateway	dc_rpc_get_gateway_address	CBLDCRPC('GETGWADR')

Facility		Library function name	
		C language library	COBOL-UAP creation program
	Report data to CUP unidirectionally	dc_rpc_cltsend	CBLDCRPC('CLTSEND ')
	Terminate a UAP	dc_rpc_close	CBLDCRPC('CLOSE ')
Remote API facility	Establish a connection with a RAP-processing listener	dc_rap_connect	CBLDCRAP('CONNECT ') CBLDCRAP('CONNECTX')
	Release a connection with a RAP-processing listener	dc_rap_disconnect	CBLDCRAP('DISCNCT ')
Transaction control	Start a transaction	dc_trn_begin	CBLDCRPC('BEGIN ')
	Enable commitment in chained mode	dc_trn_chained_commit	CBLDCRPC('C-COMMIT')
	Enable rollback in chained mode	dc_trn_chained_rollback	CBLDCRPC('C-ROLL ')
	Enable commitment in unchained mode	dc_trn_unchained_commit	CBLDCRPC('U-COMMIT')
	Enable rollback in unchained mode	dc_trn_unchained_rollback	CBLDCRPC('U-ROLL ')
	Report the information about the current transaction	dc_trn_info	CBLDCRPC('INFO ')
System operation management	Execute operation command	dc_adm_call_command	CBLDCADM('COMMAND ')
	Report completion of processing that starts a user server	dc_adm_complete	CBLDCADM('COMPLETE')
	Report the status of a user server	dc_adm_status	CBLDCADM('STATUS ')
Output audit log	Output audit log	dc_log_audit_print	CBLDCADT('PRINT ')
Output message log	Output message log	dc_logprint	CBLDCLOG('PRINT ')

Facility		Library function name	
		C language library	COBOL-UAP creation program
User journal acquisition	Acquire user journal	dc_jnl_ujput	CBLDCJNL('UJPUT ')
Journal data editing#2	Close the jnlrput output file	--	CBLDCJUP('CLOSERPT')
	Open the jnlrput output file	--	CBLDCJUP ('OPENRPT ')
	Input journal data of the jnlrput output file	--	CBLDCJUP('RDGETRPT')
Performance verification trace	Acquire user-specific performance verification traces	dc_prf_utrace_ put	CBLDCPRF('PRFPUT ')
	Report the sequential number for an acquired performance verification trace	dc_prf_get_ trace_num	CBLDCPRF('PRFGETN ')
Real-time statistical information service	Acquire real-time statistical information for arbitrary section	dc_rts_utrace_put	CBLDCRTS('RTSPUT ')

Legend:

--: Not applicable

#1

You cannot use this facility on the COBOL-UAP creation program.

#2

Journal data editing cannot use C Language API.

Table 1-2: OpenTP1 library functions (TP1/Message Control functions)

Facility		Library function name	
		C language library	COBOL-UAP creation program
Message exchanging	Open the MCF environment	dc_mcf_open	CBLDCMCF('OPEN ')
	Start an MHP service	dc_mcf_mainloop	CBLDCMCF('MAINLOOP')

Facility		Library function name	
		C language library	COBOL-UAP creation program
	Receive a message	dc_mcf_receive	CBLDCMCF('RECEIVE')
	Send a response message	dc_mcf_reply	CBLDCMCF('REPLY ')
	Send a message	dc_mcf_send	CBLDCMCF('SEND ')
	Resend a message	dc_mcf_resend	CBLDCMCF('RESEND ')
	Receive a synchronous message	dc_mcf_recvsync	CBLDCMCF('RECVSYNC')
	Send a synchronous message	dc_mcf_sendsync	CBLDCMCF('SENDSYNC')
	Exchange a synchronous message	dc_mcf_sendrecv	CBLDCMCF('SENDRECV')
	Accept temporary-stored data	dc_mcf_tempget	CBLDCMCF('TEMPGET ')
	Update temporary-stored data	dc_mcf_tempput	CBLDCMCF('TEMPPUT ')
	Terminate continuous-inquiry-response processing	dc_mcf_contend	CBLDCMCF('CONTEND ')
	Activate an application program	dc_mcf_execap	CBLDCMCF('EXECAP ')
	Report the application information	dc_mcf_ap_info	CBLDCMCF('APINFO ')
	Report the application information to user exit routines	dc_mcf_ap_info_uoc	--
	Set user timer monitoring	dc_mcf_timer_set	CBLDCMCF('TIMERSET')
	Cancel user timer monitoring	dc_mcf_timer_cancel	CBLDCMCF('TIMERCAN')
	Commit an MHP	dc_mcf_commit	CBLDCMCF('COMMIT ')
	Enable MHP rollback	dc_mcf_rollback	CBLDCMCF('ROLLBACK')
	Close the MCF environment	dc_mcf_close	CBLDCMCF('CLOSE ')

Facility		Library function name	
		C language library	COBOL-UAP creation program
	Acquire the MCF communication service status	dc_mcf_tlscom	CBLDCMCF('TLSCOM ')
	Acquire the connection status	dc_mcf_tlscn	CBLDCMCF('TLSCN ')
	Establish a connection	dc_mcf_tactcn	CBLDCMCF('TACTCN ')
	Release a connection	dc_mcf_tdctcn	CBLDCMCF('TDCTCN ')
	Acquire the acceptance status for a server-type connection establishment request	dc_mcf_tlsln	CBLDCMCF('TLSLN ')
	Start acceptance of server-type connection establishment requests	dc_mcf_tonln	CBLDCMCF('TONLN ')
	Terminate acceptance of server-type connection establishment requests	dc_mcf_tofln	CBLDCMCF('TOFLN ')
	Delete an application-related timer activation request	dc_mcf_adltap	CBLDCMCF('ADLTAP ')
	Acquire the status of a logical terminal	dc_mcf_tlsle	CBLDCMCF('TLSLE ')
	Shut down a logical terminal	dc_mcf_tdctle	CBLDCMCF('TDCTLE ')
	Release a logical terminal from shutdown	dc_mcf_tactle	CBLDCMCF('TACTLE ')
	Delete the output queue of a logical terminal	dc_mcf_tdlqle	CBLDCMCF('TDLQLE ')

Legend:

--: Not applicable

Table 1-3: OpenTP1 library functions (user data manipulation functions)

Facility		Library function name	
		C language library	COBOL-UAP creation program
DAM file service	Open a logical file	dc_dam_open	CBLDCDAM('DCDAMSVC','OPEN')
	Input a logical file block	dc_dam_read	CBLDCDAM('DCDAMSVC','READ')
	Update a logical file block	dc_dam_rewrite	CBLDCDAM('DCDAMSVC','REWT')
	Output a logical file block	dc_dam_write	CBLDCDAM('DCDAMSVC','WRIT')
	Close a logical file	dc_dam_close	CBLDCDAM('DCDAMSVC','CLOS')
	Shut down a logical file	dc_dam_hold	CBLDCDAM('DCDAMSVC','HOLD')
	Release a logical file from the shutdown state	dc_dam_release	CBLDCDAM('DCDAMSVC','RLES')
	Reference the status of a logical file	dc_dam_status	CBLDCDAM('DCDAMSVC','STAT')
	Start using an unrecoverable DAM file	dc_dam_start	CBLDCDAM('DCDAMSVC','STRT')
	Terminate using an unrecoverable DAM file	dc_dam_end	CBLDCDAM('DCDAMSVC','END ')
	Allocate a physical file	dc_dam_create	CBLDCDMB('DCDAMINT','CRAT')
	Open a physical file	dc_dam_iopen	CBLDCDMB('DCDAMINT','OPEN')
	Input a physical file block	dc_dam_get	CBLDCDMB('DCDAMINT','GET ')
	Output a physical file block	dc_dam_put	CBLDCDMB('DCDAMINT','PUT ')
	Seek a physical file block	dc_dam_bseek	CBLDCDMB('DCDAMINT','BSEK')
	Input directly a physical file block	dc_dam_dget	CBLDCDMB('DCDAMINT','DGET')

Facility		Library function name	
		C language library	COBOL-UAP creation program
	Output directly a physical file block	dc_dam_dput	CBLDCDMB('DCDAMINT','DPUT')
	Close a physical file	dc_dam_iclose	CBLDCDMB('DCDAMINT','CLOS')
TAM file service	Open a TAM table#	dc_tam_open	--
	Input a TAM table record	dc_tam_read	CBLDCTAM('FxxR')('FxxU')('VxxR')('VxxU')
	Update a TAM table record on the assumption of input	dc_tam_rewrite	CBLDCTAM ('MFY ')('MFYS') ('STR ')('WFY ') ('WFYS')('YTR ')
	Update/add a TAM table record	dc_tam_write	
	Delete a TAM table record	dc_tam_delete	CBLDCTAM('ERS ') ('ERSR')('BRS ') ('BRSR')
	Cancel the input of a TAM table record#	dc_tam_read_cancel	--
	Acquire TAM table status	dc_tam_get_inf	CBLDCTAM('GST ')
	Acquire TAM table information	dc_tam_status	CBLDCTAM('INFO')
	Close a TAM table#	dc_tam_close	--
IST service	Open an internode shared table	dc_ist_open	CBLDCIST('DCISTSVC','OPEN')
	Input an internode shared table record	dc_ist_read	CBLDCIST('DCISTSVC','READ')
	Output an internode shared table record	dc_ist_write	CBLDCIST('DCISTSVC','WRIT')
	Delete an internode shared table record	dc_ist_close	CBLDCIST('DCISTSVC','CLOS')

Facility		Library function name	
		C language library	COBOL-UAP creation program
Resource lock control	Enable locking of a resource	dc_lck_get	CBLDCLCK('GET ')
	Release all resources from lock	dc_lck_release_all	CBLDCLCK('RELALL ')
	Release the resource from lock specified by name	dc_lck_release_byname	CBLDCLCK('RELNAME ')

Legend:

--: Not applicable

#

COBOL-UAP creation programs cannot be used.

Table 1-4: OpenTP1 library functions (X/Open-compatible functions)

Facility		Library function name	
		C language library	COBOL-UAP creation program
XATMI interface	Send a service request and synchronously await its reply	tpcall()	TPCALL
	Send a service request	tpacall()	TPACALL
	Get a reply from a previous service request	tpgetrply()	TPGETRPLY
	Cancel a call descriptor for an outstanding reply	tpcancel()	TPCANCEL
	Establish a conversational service connection	tpconnect()	TPCONNECT
	Terminate a conversational service connection abortively	tpdiscon()	TPDISCON
	Receive a message in a conversational connection	tprecv()	TPRECV

Facility		Library function name	
		C language library	COBOL-UAP creation program
	Send a message in a conversational connection	tpsend()	TPSEND
	Allocate a typed buffer	tpalloc()	--
	Free a typed buffer	tpfree()	--
	Change the size of a typed buffer	tprealloc()	--
	Determine information about a typed buffer	tptypes()	--
	Advertise a service name	tpadvertise()	TPADVERTISE
	Unadvertise a service name	tpunadvertise()	TPUNADVERTISE
	Template for service routines	tpservice()	TPSVCSTART
	Return from a service routine	tpreturn()	TPRETURN
TX interface	Begin a global a transaction	tx_begin()	TXBEGIN
	Commit a global transaction	tx_commit()	TXCOMMIT
	Return global transaction information	tx_info()	TXINFORM
	Open a set of resource managers	tx_open()	TXOPEN
	Roll back a global transaction	tx_rollback()	TXROLLBACK
	Close a set of resource managers	tx_close()	TXCLOSE
	Set commit_return characteristic	tx_set_commit_return()	TXSETCOMMITRET
	Set transaction_control characteristic	tx_set_transaction_control()	TXSETTRANCTL

Facility		Library function name	
		C language library	COBOL-UAP creation program
	Set transaction_ timeout characteristic	tx_set_ transaction_ timeout0	TXSETTIMEOUT

Legend:

--: There is no COBOL API on the XATMI interface that provides this facility.

Table 1-5: OpenTP1 library functions (functions used in special style)

Facility		Library function name	
		C language library	COBOL-UAP creation program
Multinode facility#	Start acquiring the status of OpenTP1 node	dc_adm_get_nd_ status_begin	--
	Acquire the status of OpenTP1 node	dc_adm_get_nd_ status_next	--
	Acquire the status of a specified OpenTP1 node	dc_adm_get_nd_ status	--
	Terminate acquiring the status of OpenTP1 node	dc_adm_get_nd_ status_done	--
	Start acquiring a node identifiers	dc_adm_get_ nodeconf_begin	--
	Acquire a node identifier	dc_adm_get_ nodeconf_next	--
	Terminate acquiring a node identifiers	dc_adm_get_ nodeconf_done	--
	Acquire the node identifier of the local node	dc_adm_get_node_ id	--
	Start acquiring the status of user server	dc_adm_get_sv_ status_begin	--
	Acquire the status of user server	dc_adm_get_sv_ status_next	--

Facility		Library function name	
		C language library	COBOL-UAP creation program
	Acquire the status of a specified user server	dc_adm_get_sv_status	--
	Terminate acquiring the status of user server	dc_adm_get_sv_status_done	--
Online tester management	Report the test status of a user server	dc_uto_test_status	CBLDCUTO ('T-STATUS')

Legend:

--: Not applicable

#

Multinode facility APIs cannot use COBOL-UAP creation programs.

(2) Types of UAP and available library functions

Tables 1-6 to 1-10 list the library functions available to OpenTP1 UAPs. The UAPs that can use the listed library functions are SUPs, SPPs, MHPs, and UAPs that handle offline work.

Table 1-6: Library functions available with UAPs (basic OpenTP1 facilities)

OpenTP1 library function name	SUP		SPP			MHP		Off-line
	Out	In	Out	Transaction range		Out	In	
				Rt	N-Rt			
dc_rpc_open	Y	N	O	N	N	O	N	N
dc_rpc_mainloop	N	N	O	N	N	N	N	N
dc_rpc_call	Y	Y	Y	Y	Y	Y	Y	N
dc_rpc_call_to	Y	Y	Y	Y	Y	Y	Y	N
dc_rpc_poll_any_replies	Y	Y	Y	Y	Y	Y	Y	N
dc_rpc_get_error_descriptor	Y	Y	Y	Y	Y	Y	Y	N
dc_rpc_discard_further_replies	Y	Y	Y	Y	Y	Y	Y	N

OpenTP1 library function name	SUP		SPP			MHP		Off-line
	Out	In	Out	Transaction range		Out	In	
				Rt	N-Rt			
dc_rpc_discard_specific_reply	Y	Y	Y	Y	Y	Y	Y	N
dc_rpc_service_retry	N	N	Y	N	N	Y	N	N
dc_rpc_set_service_prio	Y	Y	Y	Y	Y	Y	Y	N
dc_rpc_get_service_prio	Y	Y	Y	Y	Y	Y	Y	N
dc_rpc_get_watch_time	Y	Y	Y	Y	Y	Y	Y	N
dc_rpc_set_watch_time	Y	Y	Y	Y	Y	Y	Y	N
dc_rpc_get_callers_address	N	N	Y	Y	Y	N	N	N
dc_rpc_get_gateway_address	N	N	Y	Y	Y	N	N	N
dc_rpc_cltsend	Y	Y	Y	Y	Y	Y	Y	N
dc_rpc_close	Y	N	O	N	N	O	Y	N
dc_rap_connect	Y	N	Y	N	N	Y	N	N
dc_rap_disconnect	Y	N	Y	N	N	Y	N	N
dc_trn_begin#	Y	N	Y	N	N	O	N	N
dc_trn_chained_commit#	N	Y	N	Y	N	N	N	N
dc_trn_chained_rollback#	N	Y	N	Y	N	N	N	N
dc_trn_unchained_commit#	N	Y	N	Y	N	N	O	N
dc_trn_unchained_rollback#	N	Y	N	Y	Y	N	O	N
dc_trn_info	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_call_command	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_complete	Y	N	N	N	N	N	N	N
dc_adm_status	Y	Y	Y	Y	Y	Y	Y	N
dc_log_audit_print	Y	Y	Y	Y	Y	Y	Y	N
dc_logprint	Y	Y	Y	Y	Y	Y	Y	N

OpenTP1 library function name	SUP		SPP			MHP		Off-line
	Out	In	Out	Transaction range		Out	In	
				Rt	N-Rt			
dc_jnl_ujput#	N	Y	N	Y	Y	N	Y	N
CBLDCJUP('CLOSERPT')	N	N	N	N	N	N	N	Y
CBLDCJUP('OPENRPT ')	N	N	N	N	N	N	N	Y
CBLDCJUP('RDGETRPT')	N	N	N	N	N	N	N	Y
dc_prf_utrace_put	Y	Y	Y	Y	Y	Y	Y	Y
dc_prf_trace_num	Y	Y	Y	Y	Y	Y	Y	Y
dc_rts_utrace_put	Y	Y	Y	Y	Y	Y	Y	N

Legend:

Out: Outside transaction range

In: Inside transaction range (root)

Rt: Root

N-Rt: Non-root

Off-line: UAP that handles offline work

Y: Can be used with UAPs.

O: Can be used only by the main function.

N: Cannot be used with UAPs.

Note

Outside transaction range indicates the range of nontransaction attribute MHPs or MHP main functions.

#

UAPs which use this function must be specified so that they will be run as transactions as follows:

TP1/Server Base:

- Specify `atomic_update=Y` in the user service definition.

TP1/LiNK:

- When setting up an application program environment, specify that the transaction facility will be used.

Table 1-7: Library functions available with UAPs (TP1/Message Control functions)

OpenTP1 library function name	SUP		SPP			MHP		Off-line
	Out	In	Out	Transaction range		Out	In	
				Rt	N-Rt			
dc_mcf_open	--	--	M	--	--	M	M	--
dc_mcf_mainloop	--	--	--	--	--	M	--	--
dc_mcf_receive	--	--	--	--	--	N	Y	--
dc_mcf_reply	--	--	--	--	--	N	Y	--
dc_mcf_send	--	--	--	Y	Y	N	Y	--
dc_mcf_resend	--	--	--	Y	Y	--	Y	--
dc_mcf_recvsync	--	--	Y	Y	Y	Y	Y	--
dc_mcf_sendsync	--	--	Y	Y	Y	Y	Y	--
dc_mcf_sendrecv	--	--	Y	Y	Y	Y	Y	--
dc_mcf_tempget	--	--	--	--	--	N	Y	--
dc_mcf_tempput	--	--	--	--	--	N	Y	--
dc_mcf_contend	--	--	--	--	--	N	Y	--
dc_mcf_execap	--	--	--	Y	Y	N	Y	--
dc_mcf_ap_info	--	--	--	--	--	N	Y	--
dc_mcf_ap_info_uoc	--	--	--	--	--	N	Y	--
dc_mcf_timer_set	--	--	Y	Y	Y	Y	Y	--
dc_mcf_timer_cancel	--	--	Y	Y	Y	Y	Y	--
dc_mcf_commit	--	--	--	--	--	--	Y	--
dc_mcf_rollback	--	--	--	--	--	--	Y	--
dc_mcf_close	--	--	M	--	--	M	M	--
dc_mcf_tlscom	--	--	Y	Y	Y	Y	Y	--

OpenTP1 library function name	SUP		SPP			MHP		Off-line
	Out	In	Out	Transaction range		Out	In	
				Rt	N-Rt			
dc_mcf_tlscn	--	--	Y	Y	Y	Y	Y	--
dc_mcf_tactcn	--	--	Y	Y	Y	Y	Y	--
dc_mcf_tdctcn	--	--	Y	Y	Y	Y	Y	--
dc_mcf_tlsln	--	--	Y	Y	Y	Y	Y	--
dc_mcf_tonln	--	--	Y	Y	Y	Y	Y	--
dc_mcf_tofln	--	--	Y	Y	Y	Y	Y	--
dc_mcf_adltap	--	--	Y	Y	Y	Y	Y	--
dc_mcf_tlsle	--	--	Y	Y	Y	Y	Y	--
dc_mcf_tdctle	--	--	Y	Y	Y	Y	Y	--
dc_mcf_tactle	--	--	Y	Y	Y	Y	Y	--
dc_mcf_tdlqle	--	--	Y	Y	Y	Y	Y	--

Legend:

Out: Outside transaction range

In: Inside transaction range (root)

Rt: Root

N-Rt: Non-root

Off-line: UAP that handles offline work

Y: Can be used with UAPs.

M: Can be used only by the main function.

N: Can be used only from MHPs which have the nontransaction attribute and are in the service function range.

--: Cannot be used with UAPs.

*N*ote

Outside transaction range indicates the range of nontransaction attribute MHPs or MHP main functions.

Table 1-8: Library functions available with UAPs (operate user data)

OpenTP1 library function name	SUP		SPP			MHP		Off-line
	Out	In	Out	Transaction range		Out	In	
				Rt	N-Rt			
dc_dam_open#	Y	Y	Y	Y	Y	Y	Y	N
dc_dam_read#	Y	Y	Y	Y	Y	Y	Y	N
dc_dam_rewrite#	(Y)	Y	(Y)	Y	Y	(Y)	Y	N
dc_dam_write#	(Y)	Y	(Y)	Y	Y	(Y)	Y	N
dc_dam_close#	Y	Y	Y	Y	Y	Y	Y	N
dc_dam_hold	N	Y	N	Y	Y	N	Y	N
dc_dam_release	Y	Y	Y	Y	Y	N	Y	N
dc_dam_status	Y	Y	Y	Y	Y	Y	Y	N
dc_dam_start	Y	Y	Y	Y	Y	Y	Y	N
dc_dam_end	Y	Y	Y	Y	Y	Y	Y	N
dc_dam_create	N	N	N	N	N	N	N	Y
dc_dam_iopen	N	N	N	N	N	N	N	Y
dc_dam_get	N	N	N	N	N	N	N	Y
dc_dam_put	N	N	N	N	N	N	N	Y
dc_dam_bseek	N	N	N	N	N	N	N	Y
dc_dam_dget	N	N	N	N	N	N	N	Y
dc_dam_dput	N	N	N	N	N	N	N	Y
dc_dam_iclose	N	N	N	N	N	N	N	Y
dc_tam_open	Y	Y	Y	Y	Y	Y	Y	N
dc_tam_read	N	Y	N	Y	Y	N	Y	N
dc_tam_rewrite	N	Y	N	Y	Y	N	Y	N
dc_tam_write	N	Y	N	Y	Y	N	Y	N
dc_tam_delete	N	Y	N	Y	Y	N	Y	N

OpenTP1 library function name	SUP		SPP			MHP		Off-line
	Out	In	Out	Transaction range		Out	In	
				Rt	N-Rt			
dc_tam_read_cancel	N	Y	N	Y	Y	N	Y	N
dc_tam_get_inf	Y	Y	Y	Y	Y	Y	Y	N
dc_tam_status	Y	Y	Y	Y	Y	Y	Y	N
dc_tam_close	Y	Y	Y	Y	Y	Y	Y	N
dc_ist_open	Y	Y	Y	Y	Y	Y	Y	N
dc_ist_read	Y	Y	Y	Y	Y	Y	Y	N
dc_ist_write	Y	Y	Y	Y	Y	Y	Y	N
dc_ist_close	Y	Y	Y	Y	Y	Y	Y	N
dc_lck_get#	N	Y	N	Y	Y	N	Y	N
dc_lck_release_all#	N	Y	N	Y	Y	N	Y	N
dc_lck_release_byname#	N	Y	N	Y	Y	N	Y	N

Legend:

Out: Outside transaction range

In: Inside transaction range (root)

Rt: Root

N-Rt: Non-root

Off-line: UAP that handles offline work

Y: Can be used with UAPs.

(Y): Can be used only with unrecoverable DAM files.

N: Cannot be used with UAPs.

Note

Outside transaction range indicates the range of nontransaction attribute MHPs or MHP main functions.

#

For UAPs which use this function, specify the transaction attribute (specify `atomic_update=Y` in the user service definition) for TP1/Server Base. However, transaction processing is not assumed with these UAPs when an unrecoverable DAM file is accessed.

For TP1/LiNK, any UAPs which use these functions are unavailable.

Table 1-9: Library functions available with UAPs (X/Open-compatible functions)

OpenTP1 library function name	SUP		SPP			MHP		Off-line
	Out	In	Out	Transaction range		Out	In	
				Rt	N-Rt			
`tpcall`	Y	Y	Y	Y	Y	N	N	N
`tpacall`	Y	Y	Y	Y	Y	N	N	N
`tpgetrply`	Y	Y	Y	Y	Y	N	N	N
`tpcancel`	Y	Y	Y	Y	Y	N	N	N
`tpconnect`	Y	Y	Y	Y	Y	N	N	N
`tpdiscon`	Y	Y	Y	Y	Y	N	N	N
`tprecv`	Y	Y	Y	Y	Y	N	N	N
`tpsend`	Y	Y	Y	Y	Y	N	N	N
`tpalloc`	Y	Y	Y	Y	Y	N	N	N
`tpfree`	Y	Y	Y	Y	Y	N	N	N
`tprealloc`	Y	Y	Y	Y	Y	N	N	N
`tptypes`	Y	Y	Y	Y	Y	N	N	N
`tpadvertise`	N	N	Y#3	Y#3	Y#3	N	N	N
`tpunadvertise`	N	N	Y#3	Y#3	Y#3	N	N	N
`tpservice`#1	N	N	N	N	N	N	N	N
`tpreturn`	N	N	Y#4	Y#4	Y#4	N	N	N
`tx_begin`#2	Y	Y	Y	N	N	Y	N	N

OpenTP1 library function name	SUP		SPP			MHP		Off-line
	Out	In	Out	Transaction range		Out	In	
				Rt	N-Rt			
tx_commit with TX_CHAINED#2	N	Y	Y	N	N	N	N	N
tx_commit with TX_UNCHAINED#2	N	Y	Y	N	N	N	N	N
tx_info	Y	Y	Y	Y	Y	N	N	N
tx_open	Y	N	Y	N	N	N	N	N
tx_rollback with TX_CHAINED#2	N	Y	N	Y	N	N	N	N
tx_rollback with TX_UNCHAINED#2	N	Y	N	Y	Y	N	N	N
tx_close	Y	N	Y	N	N	N	N	N
tx_set_commit_return#2	Y	Y	Y	Y	Y	N	N	N
tx_set_transaction_control #2	Y	Y	Y	Y	Y	N	N	N
tx_set_transaction_timeout #2	Y	Y	Y	Y	Y	N	N	N

Legend:

Out: Outside transaction range

In: Inside transaction range (root)

Rt: Root

N-Rt: Non-root

Off-line: UAP that handles offline work

Y: Can be used with UAPs.

N: Cannot be used with UAPs.

#1

tpservice is the entity of the service function.

#2

UAPs which use this function must be specified so that they will be run as transactions as follows:

TP1/Server Base:

- Specify `atomic_update=Y` in the user service definition.

TP1/LiNK:

- When setting up an application program environment, specify that the transaction facility will be used.

#3

This function can be called only within service functions.

#4

This function is used only to make XATMI-interfaced service functions return.

Table 1-10: Library functions available with UAPs (functions used in special style)

OpenTP1 library function name	SUP		SPP			MHP		Off-line
	Out	In	Out	Transaction range		Out	In	
				Rt	N-Rt			
dc_adm_get_nd_status_begin#	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_get_nd_status_next#	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_get_nd_status#	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_get_nd_status_done#	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_get_nodeconf_begin#	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_get_nodeconf_next#	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_get_nodeconf_done#	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_get_node_id#	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_get_sv_status_begin	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_get_sv_status_next	Y	Y	Y	Y	Y	Y	Y	N
dc_adm_get_sv_status	Y	Y	Y	Y	Y	Y	Y	N

OpenTP1 library function name	SUP		SPP			MHP		Off-line
	Out	In	Out	Transaction range		Out	In	
				Rt	N-Rt			
dc_adm_get_sv_status_done	Y	Y	Y	Y	Y	Y	Y	N
dc_uto_test_status	Y	Y	Y	Y	Y	Y	Y	N

Legend:

Out: Outside transaction range

In: Inside transaction range (root)

Rt: Root

N-Rt: Non-root

Off-line: UAP that handles offline work

Y: Can be used with UAPs.

N: Cannot be used with UAPs.

#

Before a node can use functions marked #, TP1/Multi must be installed in the node.

1.5 Debuggers and testers for application programs

OpenTP1 allows created UAPs to be tested for operation before they are put into use for actual jobs. The facilities for testing UAPs is referred to as the *UAP tester facilities*.

The UAP tester facilities eliminate the need for modifying actually used resources for test purposes. In addition, these facilities provide tests which are conducted in response to commands entered by the operator so that the operator can check for important test items before starting tests.

1.5.1 Types of UAP tester facility

The following OpenTP1 UAP tester facilities are available, each serving a different purpose.

(1) Offline tester (TP1/Offline Tester)

The offline tester tests online processing UAPs in an offline environment. It tests the UAP for operation without having to run OpenTP1. It is used when testing single UAPs before using them with OpenTP1 resources. Since the offline tester can be combined with debuggers written in high-level languages (C and COBOL), the program can be double-checked through UAP testing and debugging.

The offline tester tests SPPs and MHPs for operation.

Before the offline tester can be run on a machine, TP1/Offline Tester must be installed in the machine.

(2) Online tester (TP1/Online Tester)

The online tester tests UAPs in an online environment. It can test UAP operation in cooperation with OpenTP1 system services. It is used when testing OpenTP1 UAPs as integrated. Since the online tester can be combined with debuggers written in high-level languages (C and COBOL), the program can be double-checked through UAP testing and debugging.

The online tester can test SUPs and SPPs for operation. It can also test MHPs as operating like SPPs.

Before the online tester can be run on a machine, TP1/Online Tester must be installed in the machine.

(3) MCF online tester (TP1/Message Control/Tester)

The MCF online tester tests UAPs in an online environment. It is used when testing MHPs in cooperation with TP1/Message Control. UAPs can be tested using OpenTP1 system services and MCF system services.

Before the MCF online tester can be run on a machine, TP1/Message Control/Tester

must be installed in the machine. Before the UAP trace facility of the online tester can be used, TP1/Online Tester must be installed.

1.5.2 UAPs that can be tested

The UAPs that can be tested by the UAP tester facilities are SUPs, SPPs, and MHPs. The purpose of testing varies depending on the particular type of UAP tester facility.

If a CUP, which is a UAP used with the OpenTP1 client facility (TP1/Client), requests an SPP for service, it can start the SPP in test mode.

UAPs that handle offline works cannot be tested by the UAP tester facilities.

For details on the UAP tester facilities, see the *OpenTP1 Tester and UAP Trace User's Guide*.

1.5.3 Reporting the test status of a user server

When the online tester (TP1/Online Tester) is used under OpenTP1, the status of a user server can be detected. To detect the test status, use the function `dc_uto_test_status()`[`CBLDCUTO('T-STATUS')`]. This function provides the following information:

- Test user ID (the value set in the environment variable `DCUTOKEY`)
- Whether the user server runs in test mode
- Processing status of global transaction
- The following settings in the user service definition:

  Test type specified in the operand `test_mode`

  Handling of a synchronization point of the transaction specified in the operand `test_transaction_commit`

  Handling of the results of executing the command specified in the operand `test_adm_call_command`

Chapter

2. Basic OpenTP1 Facilities (TP1/ Server Base, TP1/LiNK)

This chapter explains application program facilities available at nodes where TP1/ Server Base or TP1/LiNK is used.

The facilities are explained using C-language function names. For each function, the name of the equivalent COBOL-language API function is indicated in brackets [ ] when the function appears first in this chapter. After that, only the C-language function name is written. If the C-language function has no COBOL counterpart API function, brackets are not written.

This chapter contains the following sections:

2.1 Remote procedure call
2.2 Remote API facility
2.3 Transaction control
2.4 System operation management
2.5 Message log output
2.6 Audit log output
2.7 User journal acquisition
2.8 Journal data editing
2.9 Receiving message log notification
2.10 Client/server mode communication using OSI TP
2.11 Acquiring performance verification traces
2.12 Real-time statistical information acquisition

2.1 Remote procedure call

Like a function call, OpenTP1 UAP can request a service to another UAP without recognizing which network node includes the service providing UAP. This interprocess communication is called *remote procedure call (RPC)*. There are three RPCs applicable to OpenTP1 UAPs:

- OpenTP1 specific interface
- XATMI interface (RPCs conforming X/Open specifications)
- TxRPC interface (RPCs conforming X/Open specifications)

When TCP/IP is used as the communication protocol, the above three types of remote procedure calls can be used. When OSI TP is used as the communication protocol, only the XATMI interface can be used. For details on the remote procedure calls available with OSI TP, see *2.10 Client/server mode communication using OSI TP* and *5.1 XATMI interface (client/server-mode communication)*.

This section explains OpenTP1 specific interface RPCs. For details on XATMI interface and TxRPC interface, see *5.1 XATMI interface (client/server-mode communication)* and *6.1 Communication through TxRPC interface*.

Note

Assume that you want to perform a transactional RPC on an OpenTP1 system other than the domain specified in the `all_node` clause of the system common definition. In this case, you must ensure that the node identifiers (`node_id` clause of the system common definition) of all OpenTP1 systems in the local domain and remote domain are unique. In addition, all the OpenTP1 systems must be version 03-02 or later. If these conditions are not met, the transaction may not recover properly.

2.1.1 How to implement the remote procedure call

You can request an SPP service by calling the service request function from a client UAP. To request a service from a UAP, call the function `dc_rpc_call()` [`CBLDCRPC('CALL  ')`] with the service group name# and service name specified as arguments. The service to be requested can be at the node of the client UAP or at a different node. There is no need for UAPs to consider whether the service to be requested is at the node of the client UAP. This is because the OpenTP1 name service is responsible for recognizing the node at which the service to be requested exists.

#

When a service group name is specified with domain qualification, a service request can also be addressed to the server UAP in the specified domain. For details on service requests with domain qualification, see *2.1.18 Service request*

with domain qualification.

The server UAP used by OpenTP1 is a service providing program (SPP). Client UAPs which can request SPP services are SUPs, SPPs, and MHPs.

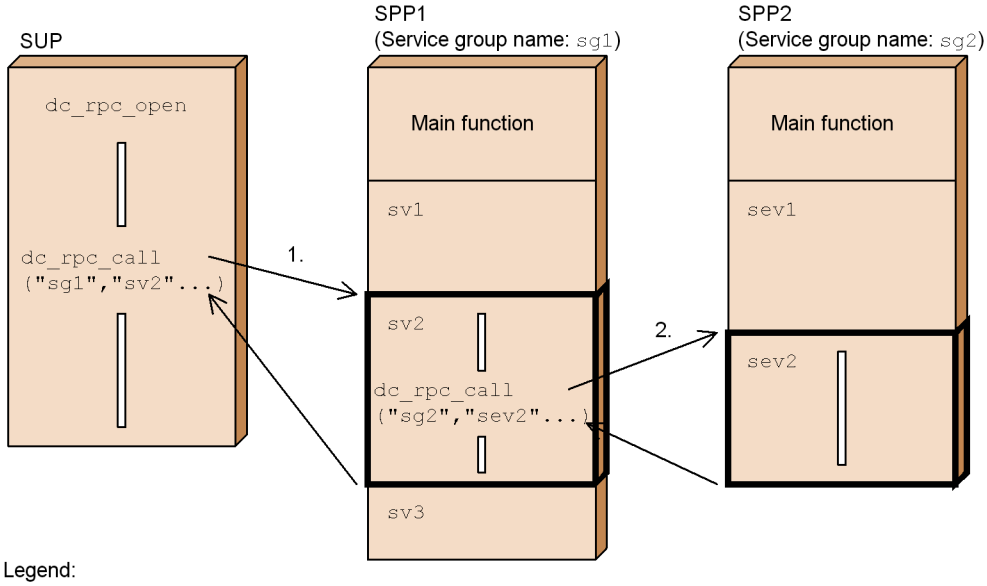
A server UAP can be started either at the same time as OpenTP1 (automatic startup) or by executing the `dcsvstart` command after OpenTP1 started (manual startup). Once started, a server UAP is ready to offer service. If a service is requested to a server UAP which is not started, the function `dc_rpc_call()` returns with an error.

A client UAP can request service using the function `dc_rpc_call()` regardless of whether the process of the started server UAP is operating. Even if the process of the server UAP specified in the service request is inactive, OpenTP1 automatically starts the process.

The MHP can request services by using the RPC. However, it cannot request MHP service functions to provide services. UAP that handles offline work cannot use the RPC.

The figure below shows the server-client relationship in communication using RPC.

Figure 2-1: Client/server relationship in communication using RPC



Legend:

1. Requests a service that has service group name `sg1` and service name `sv2` from the client UAP.
2. From service `sv2` of SPP1, to which service is requested, a service that has service group name `sg2` and service name `sev2` can be requested.

2.1.2 Transferring data through the remote procedure call

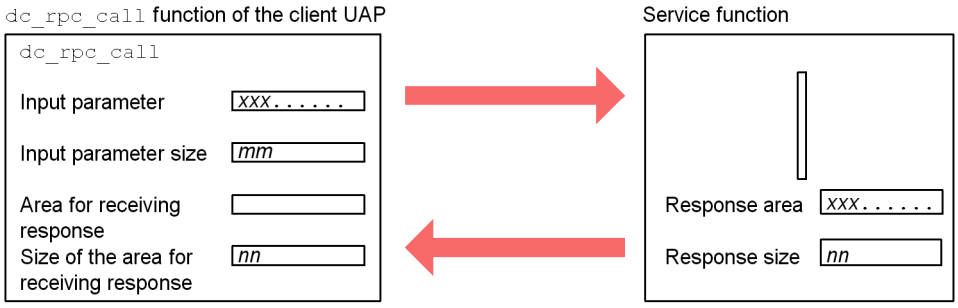
To request a service from the client UAP, specify the following as the arguments of the

function `dc_rpc_call()`:

- Input parameter
- Input parameter length
- Response storage area
- Length of the response acceptance area

The service function sets the response into the response storage area, and the response length into the response acceptance area, then returns the set values to the client UAP. The figure below shows data transfer through the remote procedure call.

Figure 2-2: Data transfer through remote procedure call

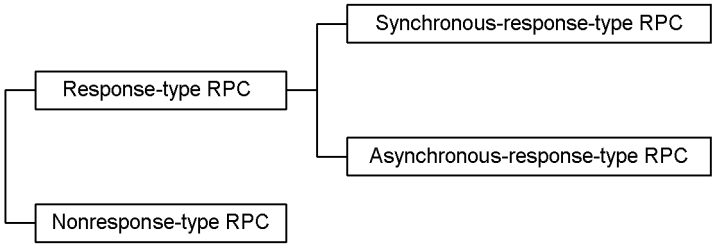


Note: The size of the response to be returned from the service function (*NN* value) must be specified as equal to or less than the value specified by the `dc_rpc_call` function (*nn* value).

2.1.3 Outline of remote procedure call modes

The RPC modes shown below are available. Set an RPC mode with the flag in the function `dc_rpc_call()` of the client UAP. The figure below shows the RPC modes.

Figure 2-3: RPC modes



- Response-type RPC

  This RPC returns the processing results of the server UAP to the client UAP. There are two types of response-type RPCs: a synchronous-response-type RPC

which waits for the processing results of the server UAP after the function `dc_rpc_call()` is called, and an asynchronous-response-type RPC which receives the processing results asynchronously.

- Nonresponse-type RPC

  This RPC does not return the processing results of the server UAP to the client UAP. After the function `dc_rpc_call()` is called, this RPC returns and continues processing. The client UAP cannot receive the processing results of the server UAP.

For details on the relationship between the synchronization point and RPC when the RPC is used for transaction processing, see *2.3.4 Relationship between remote procedure call modes and synchronization points*.

(1) Synchronous-response-type RPC

After the function `dc_rpc_call()` is called, a synchronous-response-type RPC waits for the processing results of the server UAP. To use the synchronous-response-type RPC, set `DCNOFLAGS` (or `DCRPC_CHAINED`) into `flags` of the function `dc_rpc_call()`.

(a) Time monitoring of a synchronous-response-type RPC

The response time after using the function `dc_rpc_call()` is monitored in one of the following values:

- TP1/Server Base:

  Value specified for `watch_time` in the user service definition

- TP1/LiNK:

  180 minutes

If a long server UAP processing time prevents the response from being returned within the specified monitoring time, the function `dc_rpc_call()` returns with an error.

The server UAP is aware of the response wait time on the client UAP. Therefore, when the function `dc_rpc_call()` returns with an error, the server UAP discards services that were scheduled after timeout occurred on the client UAP and does not process them. It also aborts processing of the service that was being run and does not return a response. You can instruct the server UAP to output a message indicating that the service request was discarded due to timeout on the client UAP. To do this, specify set `rpc_extend_function=00000008` in the user service definition for the server UAP.

If the server UAP terminates abnormally, the function `dc_rpc_call()` returns with an error immediately. In any of the following cases, this function returns with an error after the specified monitoring time has elapsed:

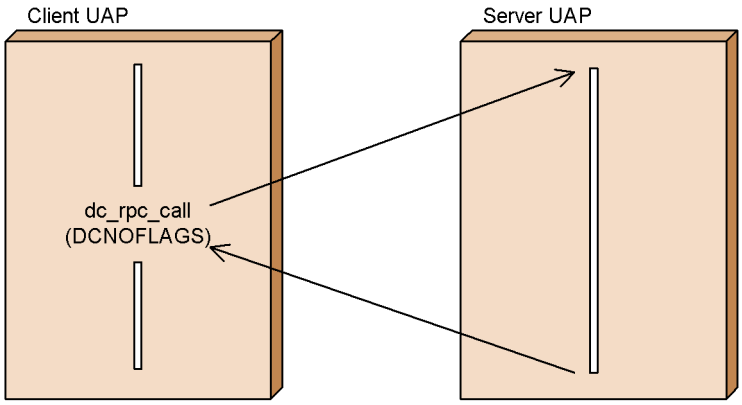
- If the entire portion of OpenTP1 for the node at which the server UAP exists

terminates abnormally

- If an error occurs before the server UAP receives service request data, or after the server UAP completes processing and before the client UAP receives the processing results

The figure below shows the synchronous-response-type RPC.

Figure 2-4: Synchronous-response-type RPC



(2) Asynchronous-response-type RPC

After the function `dc_rpc_call()` is called, an asynchronous-response-type RPC continues processing without waiting for the processing results of the server UAP. RPC parallel processing can be executed by using more than one asynchronous-response-type RPC.

The function `dc_rpc_call()` (`DCRPC_NOWAIT` set in `flags`) of an asynchronous-response-type RPC returns after requesting a service. The UAP proceeds to processing after the function returns.

An asynchronous-response-type RPC returns without checking the acceptance of the service request by the server. When a client UAP requests services through two or more asynchronous-response-type RPCs to a service group, the server may not accept the services in the order which these RPCs were issued.

(a) Receiving asynchronous-response-type RPC response

The asynchronous-response-type RPC receives the processing results of the server UAP asynchronously by using the function `dc_rpc_poll_any_replies()` [`CBLDCRPC('POLLANYR')`]. The processing results can be received only when the function `dc_rpc_poll_any_replies()` is called.

When asynchronous-response-type RPC responses are being received, a particular response to be received can be identified. To identify a response, the positive integer

(descriptor) returned by the function dc_rpc_call() for the request of the service through asynchronous-response-type RPC must be specified as the argument to the function dc_rpc_poll_any_replies(). With this specification, an asynchronous-response-type RPC response identified by the descriptor is received.

If no particular response is identified, the responses are received in the order of their returns. When the function dc_rpc_poll_any_replies() returns normally if no response is identified, the same value as the descriptor of the received asynchronous response is returned.

An error is returned if the call count of the function dc_rpc_poll_any_replies() is greater than the call count of the function dc_rpc_call() of the asynchronous-response-type RPC.

If an error occurs upon a service request, an error is returned to the function dc_rpc_poll_any_replies().

Assume that you call the function dc_rpc_call() for which DCRPC_NOWAIT is set in flags and then you call a function that executes transaction synchronization point processing before the response to the function dc_rpc_call() has been received. In such a case, when you subsequently call the function dc_rpc_poll_any_replies(), it returns with the error DCRPCER_ALL_RECEIVED and no response can be received. When an asynchronous-response type RPC is used, the response must be received before you execute transaction synchronization point processing for that process, regardless of whether or not the RPC was executed within the transaction.

(b) Time monitoring of an asynchronous-response-type RPC

When an asynchronous-response-type RPC is used, the value specified in watch_time of the user service definition is not referenced. For the argument timeout, specify the maximum response wait time from when the function dc_rpc_poll_any_replies() is called to when a response is returned.

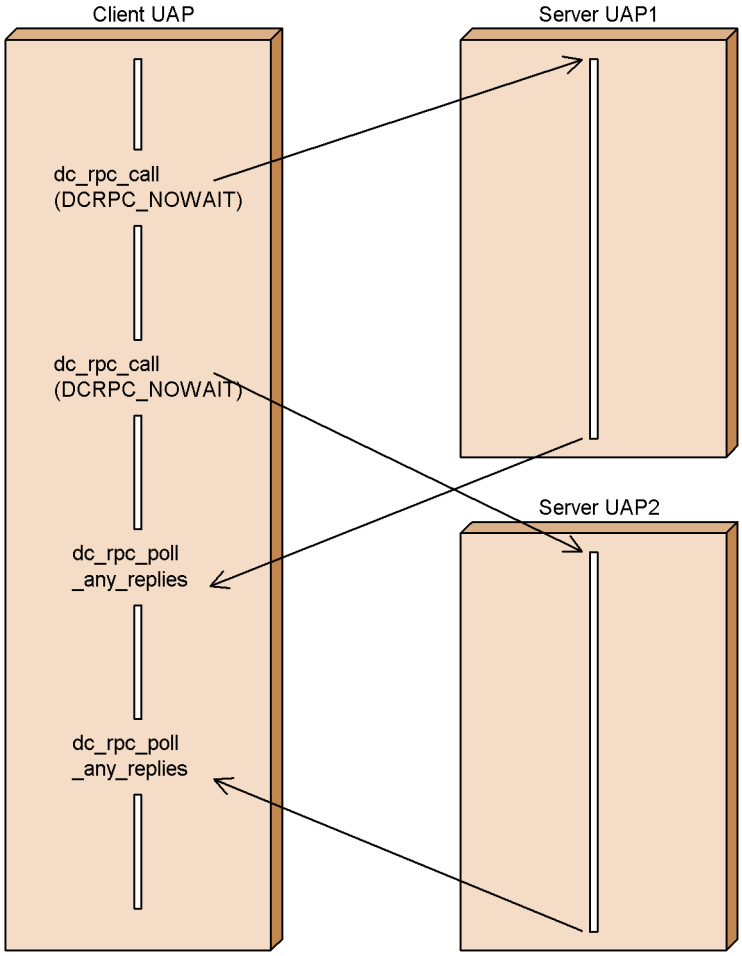
If a long server UAP processing time prevents the response from being returned within the specified monitoring time, the function dc_rpc_poll_any_replies() returns with an error.

If the server UAP terminates abnormally, the function dc_rpc_poll_any_replies() returns with an error immediately. In any of the following cases, this function returns with an error after the specified monitoring time has elapsed:

- If the entire portion of OpenTP1 for the node at which the server UAP exists terminates abnormally
- If an error occurs before the server UAP receives service request data, or after the server UAP completes processing and before the client UAP receives the processing results

The figure below shows the asynchronous receiving of processing results.

Figure 2-5: Asynchronous-response-type RPC (asynchronous receiving of processing results)



(c) Rejecting the receiving of processing results

If you do not want to receive any more replies (which have not been returned) when an asynchronous-response-type RPC is used, call the function `dc_rpc_discard_further_replies()` [`CBLDCRPC('DISCARDF')`] for rejecting the receiving of processing results. Replies returned after this function is called are discarded instead of being received. Call this function for rejecting the receiving of processing results when not receiving the results of an asynchronous-response-type RPC. Otherwise, the function `dc_rpc_poll_any_replies()` of another asynchronous-response-type RPC might receives unnecessary replies.

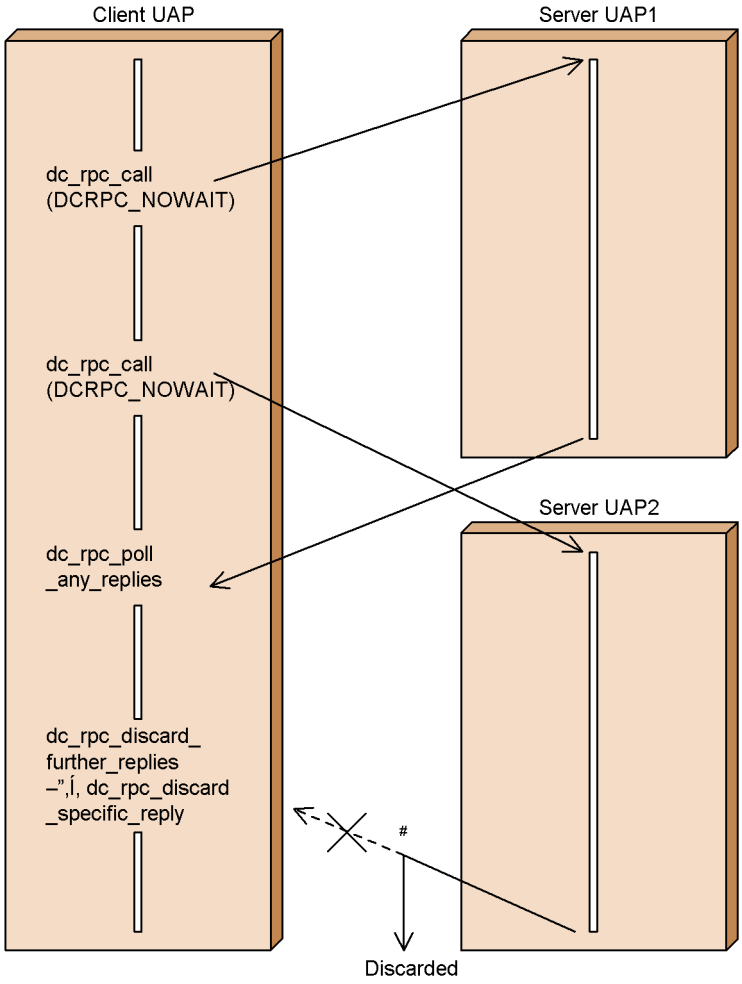
Use the function `dc_rpc_discard_further_replies()` in the following cases:

- After a response wait timeout occurs, you want to release the buffer for shutting down results before proceeding to the next processing.
- You want only the first response when having issued more than one asynchronous-response-type RPCs.

If you do not want to receive a selected response among responses which have not been returned when an asynchronous-response-type RPC is used, call the function `dc_rpc_discard_specific_reply()` [`CBLDCRPC('DISCARDS')`] for rejecting reception of selected processing results. Responses which have the same descriptor as the specified descriptor among the responses returned after this function is called are discarded instead of being received.

The figure below shows the rejection of receiving processing results.

Figure 2-6: Asynchronous-response-type RPC (rejection of receiving processing results)



Processing results returned after the function dc_rpc_discard_further_replies() is called are discarded instead of being received.
When the function dc_rpc_discard_specific_reply is called, responses which have the same descriptor as the specified descriptor are discarded.

(d) Relationship between an asynchronous-response-type RPC and a synchronization point

If an asynchronous-response-type RPC is called in a transaction, replies cannot be received asynchronously after synchronization point processing is executed. For details on the relationship between a synchronization point and an

asynchronous-response-type RPC, see *2.3.4 Relationship between remote procedure call modes and synchronization points*.

(3) Nonresponse-type RPC

A nonresponse-type RPC does not return the processing results of the server UAP to the client UAP. The client UAP cannot receive the processing results of the server UAP.

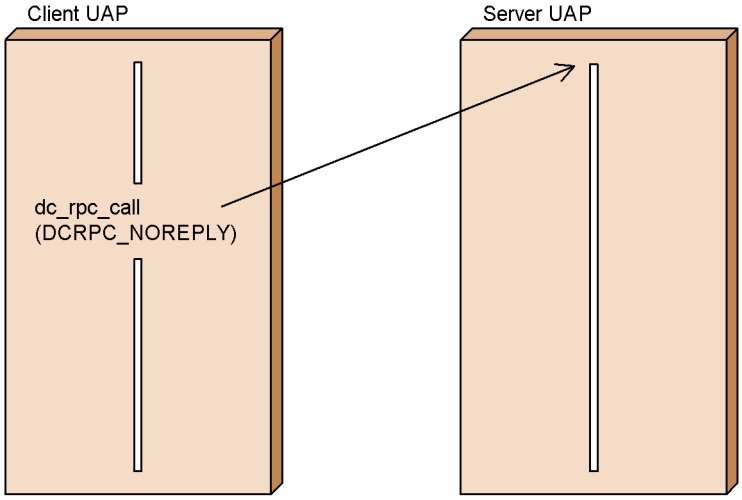
Nonresponse-type RPC time is not monitored.

The function `dc_rpc_call()` (`DCRPC_NOREPLY` set in `flags`) of the nonresponse-type RPC returns after requesting a service. The UAP proceeds to processing after the function returns. The processing results of the server UAP cannot be received.

A nonresponse-type RPC returns without checking the acceptance of the service request by the server. Therefore, if the service request is lost because of an error (e.g., communication failure), the client UAP cannot recognize this. When a client UAP requests services through two or more nonresponse-type RPCs to a service group, the server may not accept the services in the order which these RPCs were issued.

The figure below shows the nonresponse-type RPC.

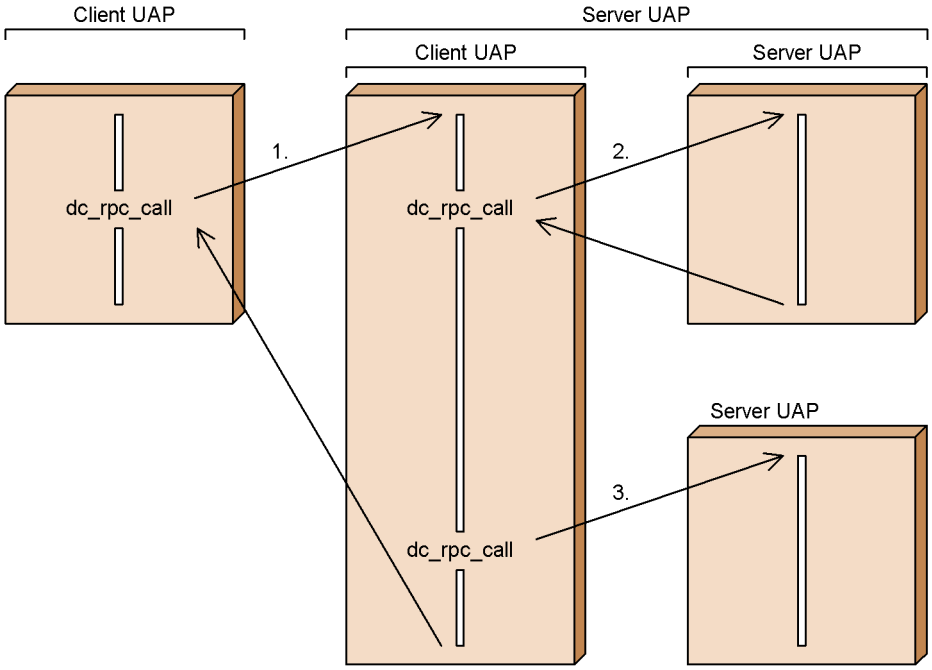
Figure 2-7: Nonresponse-type RPC



2.1.4 Nesting services

A server UAP to which a client UAP requested a service can call another server UAP. Service processing can be distributed/layered by nesting server UAPs. The figure below shows an example of nesting RPCs.

Figure 2-8: Example of nesting RPCs



1. Synchronous-response-type RPC
2. Nesting a synchronous-response-type RPC
3. Nesting a nonresponse-type RPC

2.1.5 Using nontransactional RPC from transaction process

If service is requested from a transaction process and the UAP requested for service has the transaction attribute, processing for the service request is transaction processing. Such service requests can be handled through nontransactional processing (nontransactional RPC). To effect nontransactional processing, specify the argument to the function `dc_rpc_call()` to indicate a nontransactional RPC.

For details on the transaction attribute, see *2.3.3 Specification of transaction attribute*.

2.1.6 Setting schedule priorities for service requests

Multiple service requests called for one process can be given priorities. This can be done by using the function `dc_rpc_set_service_prio()` [`CBLDCRPC ('SETSVPRI')`] in which a priority is specified for a service request before using the function `dc_rpc_call()`. The priority of the service request is then reported to the server via the schedule queue for the server UAP.

For a process that does not use the function dc_rpc_set_service_prio(), the value 4, which is the default interpretation of the schedule service, is set as the priority of service requests. The specified schedule priorities can be referenced using the function dc_rpc_get_service_prio() [CBLDCRPC('GETSVPRI')].

On queue-receiving servers (SPPs scheduled by the schedule service), the priorities specified for service requests are valid only when service_priority_control=Y (priority control in effect) is specified in the user service definition of the server UAP. If priority control is not used on the server UAP requested for service, the call of this function has no effect.

On servers that receive requests from socket (SPPs which receive service requests without intervention of a schedule queue), the priorities set by the client UAP are always in effect.

The call of the function dc_rpc_set_service_prio() for the following service requests has no effect:

- Second or subsequent service request on chained RPC
- Service request specified by the function dc_rpc_call() (DCNOFLAGS specified for flags) of synchronous-response-type RPC which is called to terminate chained RPCs.

2.1.7 Acquiring node address of client UAP

In some cases, it is desirable to limit services offered to a client UAP. Since the server UAP recognizes client UAPs, it is possible to acquire the address of the node where the process of the client UAP is operating. The node address of the client UAP can be acquired by the function dc_rpc_get_callers_address() [CBLDCRPC('GETCLADR')].

The address returned by the function dc_rpc_get_callers_address() cannot be used for sending a response to a service or a response to an error.

The function dc_rpc_get_callers_address() must be called from a service function. Otherwise, subsequent processing is unpredictable.

2.1.8 Referencing and changing response waiting intervals of service request

During UAP processing, response waiting intervals of service request can be temporarily changed. Use the function dc_rpc_get_watch_time() [CBLDCRPC('GETWATCH')] to reference the current response waiting interval and the function dc_rpc_set_watch_time() [CBLDCRPC('SETWATCH')] to change it. The value set by the function dc_rpc_set_watch_time() will be effective until the UAP calls the function dc_rpc_close().

The function dc_rpc_watch_time() returns the response waiting interval set by the function dc_rpc_set_watch_time(). If no new interval has been set, the following

value is returned:

- TP1/Server Base:

  Value given to `watch_time` in the user service definition

- TP1/LiNK:

  180 minutes

The value obtained by this function is effective as the response waiting interval for the function `dc_rpc_call()` of OpenTP1.

To return the response waiting interval of service request to the value which stood before the function `dc_rpc_set_watch_time()` was called, set in this function the original value returned by the function `dc_rpc_get_watch_time()` which was called previously.

The function `dc_rpc_set_watch_time()` influences called UAP service requests, but does not affect the value given to the `watch_time` operand in the system common definition. The value specified in this function influences only the function `dc_rpc_call()` which will be called later.

2.1.9 Acquiring descriptor of asynchronous-response-type RPC request which has encountered error

You can use a means to acquire the descriptor of an asynchronous-response-type RPC request which has encountered an error, by invoking that means just after the function `dc_rpc_poll_any_replies()` [`CBLDCRPC('POLLANYR')`] without a particular asynchronous response specified returns with an error.

The means used to acquire the descriptor of an asynchronous-response-type RPC request which has encountered an error is the function `dc_rpc_get_error_descriptor()` [`CBLDCRPC('GETERDES')`].

The descriptor of an asynchronous response can be acquired only when the error has occurred on the SPP. If an error has occurred on the `dc_rpc_poll_any_replies` [`CBLDCRPC('POLLANYR')`] caller, the function `dc_rpc_get_error_descriptor()` [`CBLDCRPC('GETERDES')`] cannot acquire the descriptor of that asynchronous response.

2.1.10 Report data to CUP unidirectionally

An OpenTP1 server UAP can report its activation to the TP1/Client application program (CUP) by using the function `dc_rpc_cltsend()` [`CBLDCRPC ('CLTSEND')`], which sends data to the CUP. This function can be used to report the activation of the server to the client simultaneously.

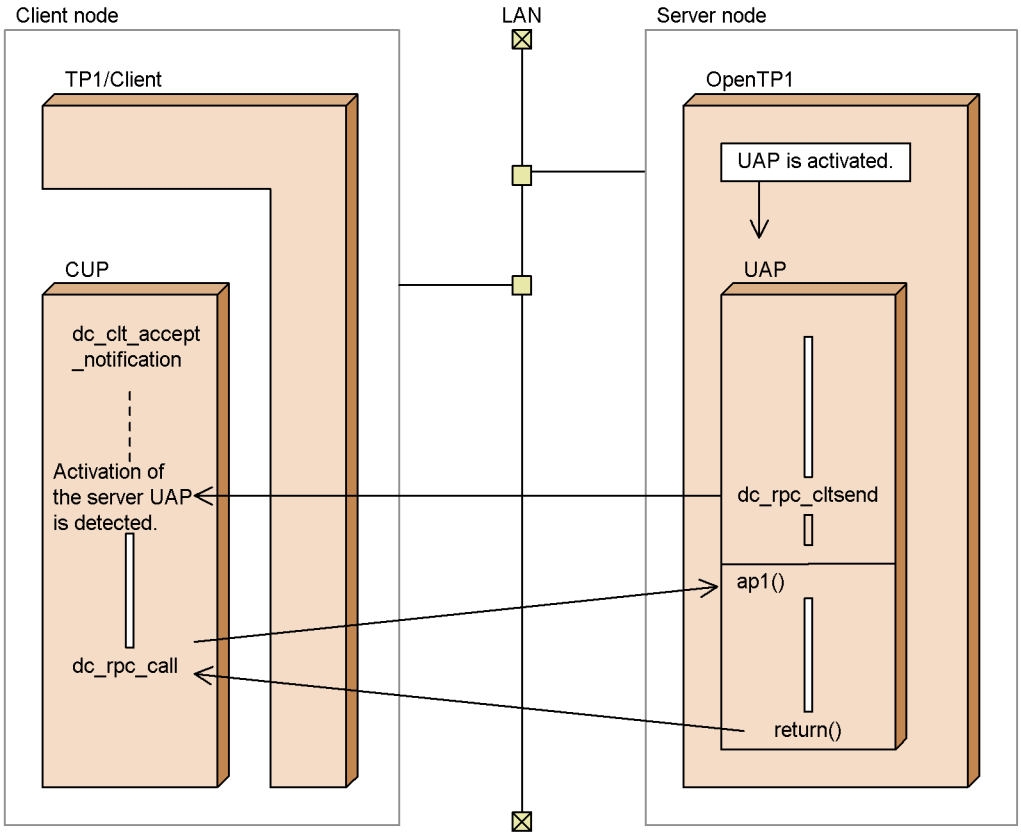
Data sent by the function `dc_rpc_cltsend()` is received with the function `dc_clt_chained_accept_notification()` or `dc_clt_accept_notification()` in the CUP. When the CUP receives data, the

TP1/Client knows the server is in operation. Then, the CUP requests a service to the server.

The function `dc_rpc_cltsend()` can be used only when the function `dc_clt_chained_accept_notification()` or `dc_clt_accept_notification()` in the CUP on the receiving end is waiting for a notification. If the CUP is not active, the function `dc_rpc_cltsend()` returns with an error. The function `dc_rpc_cltsend()` cannot send data to any processes (processes of the server UAP) other than the functions `dc_clt_chained_accept_notification()` and `dc_clt_accept_notification()` in the CUP. For details about the functions `dc_clt_chained_accept_notification()` and `dc_clt_accept_notification()`, see the manual *OpenTP1 TP1/Client User's Guide TP1/Client/W, TP1/Client/P*.

The figure below shows the outline of reporting data to CUP unidirectionally.

Figure 2-9: Outline of reporting data to CUP unidirectionally



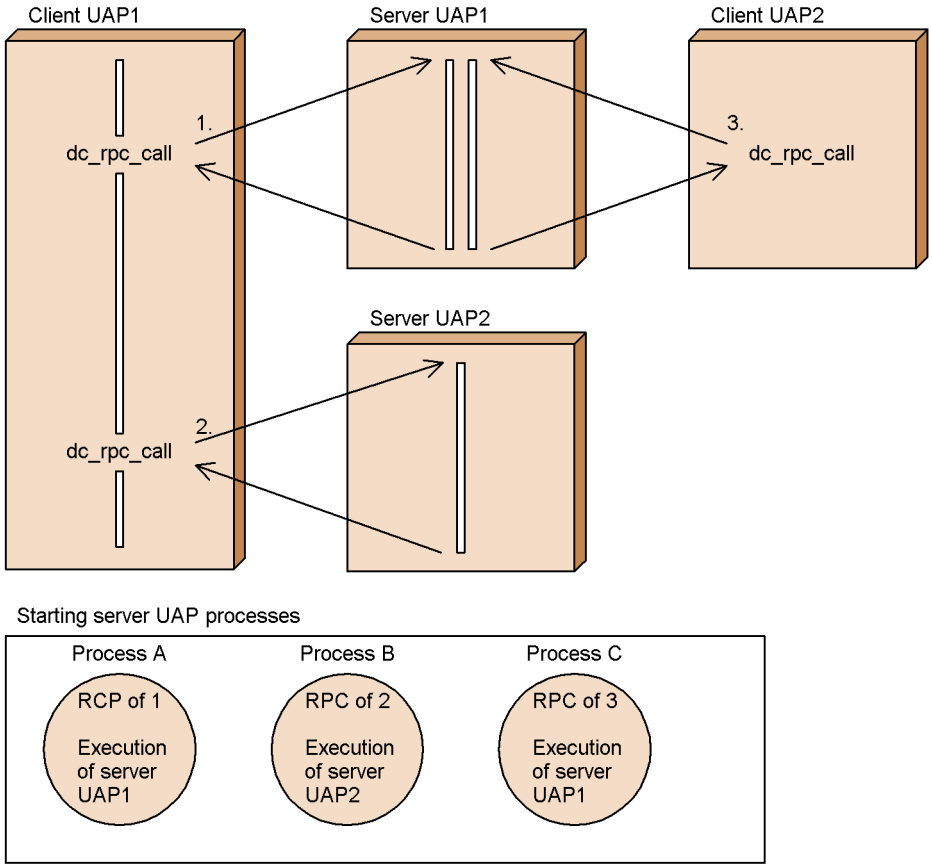
2.1.11 Relationship between remote procedure calls and processes for executing services

A server UAP to which a service request was issued is executed in a process different from the process of the client UAP. OpenTP1 can implement a multiserver that starts multiple processes for executing a server UAP. When the multiserver is used to nest RPCs, the same number of processes as services to be nested might be executed.

Even when the same server UAP is used, the server UAP is not always executed in the same process if a different client UAP is used. Also, when a service which belongs to the same service group as for the client UAP is requested, a new process is necessary for executing the service group. When using a multiserver, specify a sufficient number of processes.

The figure below shows the relationship between RPCs and processes.

Figure 2-10: Relationship between RPCs and processes



1. If a service is requested from client UAP1 to server UAP1, server UAP1 is executed in process A.
2. If a service is requested from client UAP1 to server UAP2, server UAP2 is executed in process B.
3. If a service is requested from new client UAP2 to server UAP1, server UAP1 is executed in process C unlike when a service is requested in 1.

(1) Chained RPCs

If a synchronous-response-type RPC requests the services belonging to the same service group more than once, the service offered in response to each request can be run by the same process. This RPC is called a *chained RPC*. If chained RPCs are used to request for services, the services are run by the same process as with the preceding RPC, even in a multiserver environment. The number of processes needed for transaction processing can thus be minimized. Since a UAP process is assigned for each service group, chained RPCs can be used to request for different types of service if these services belong to the same service group.

Processing for chained RPCs may or may not be performed as a transaction. If chained RPCs are run as a transaction, the transaction will be a global transaction.

Operation of a chained RPC is guaranteed within each process of the client UAP. Suppose that a service is called multiple times within the same global transaction, but that the client UAPs involved are different. It is not guaranteed that the service offered in response to each request is run by the same process.

(a) Starting chained RPCs

When requesting a service which will be the first of chained RPCs, specify `DCRPC_CHAINED` for `flags` in the function `dc_rpc_call()` that requests the service. The server UAP acquires a process with the recognition that the RPC is a chained RPC. Specify `DCRPC_CHAINED` also for `flags` in the second and subsequent service requests.

The second and subsequent service requests cannot detect the shutdown status of a user server or service. If an error occurs during execution of the second or a subsequent service request, the user server shuts down. In this situation, you cannot shut down individual services.

(b) Terminating chained RPCs

Use one of the following methods to terminate chained RPCs:

- If `00000002` is not assigned to the user service definition `rpc_extend_function` operand, issue a service request to the service groups running the chained RPCs through a synchronous-response-type RPC (with `DCNOFLAGS` assigned to `flags`).
- If the chained RPCs were initiated during a transaction, complete the global

transaction running the chained RPCs through synchronous point processing (commit or rollback).

- If `00000002` is assigned to the user service definition `rpc_extend_function` operand, issue a service request to the service groups running the chained RPCs through a synchronous-response-type RPC (with `DCNOFLAGS` assigned to `flags`) after the non-transactional chained RPCs initiated during a transaction perform synchronous point processing.

(c) Time monitoring for chained RPCs

If an SPP fails to accept a service request because of a communication error or other condition when a chained RPC is being processed, the process could remain assigned to the SPP. To avoid this, the OpenTP1 time-monitors SPPs which are activated by chained RPCs. Actually, checking is made to see whether the following value is exceeded by the period (maximum interval) from the time a response is issued to the time the next service request or a chained RPC termination request comes:

- TP1/Server Base:

  Value given to the `watch_next_chain_time` operand in the user service definition

- TP1/LiNK:

  180 seconds

If the next service request or chained RPC termination request does not come within the above interval, the OpenTP1 considers that a failure has occurred in the client UAP and aborts the SPP process.

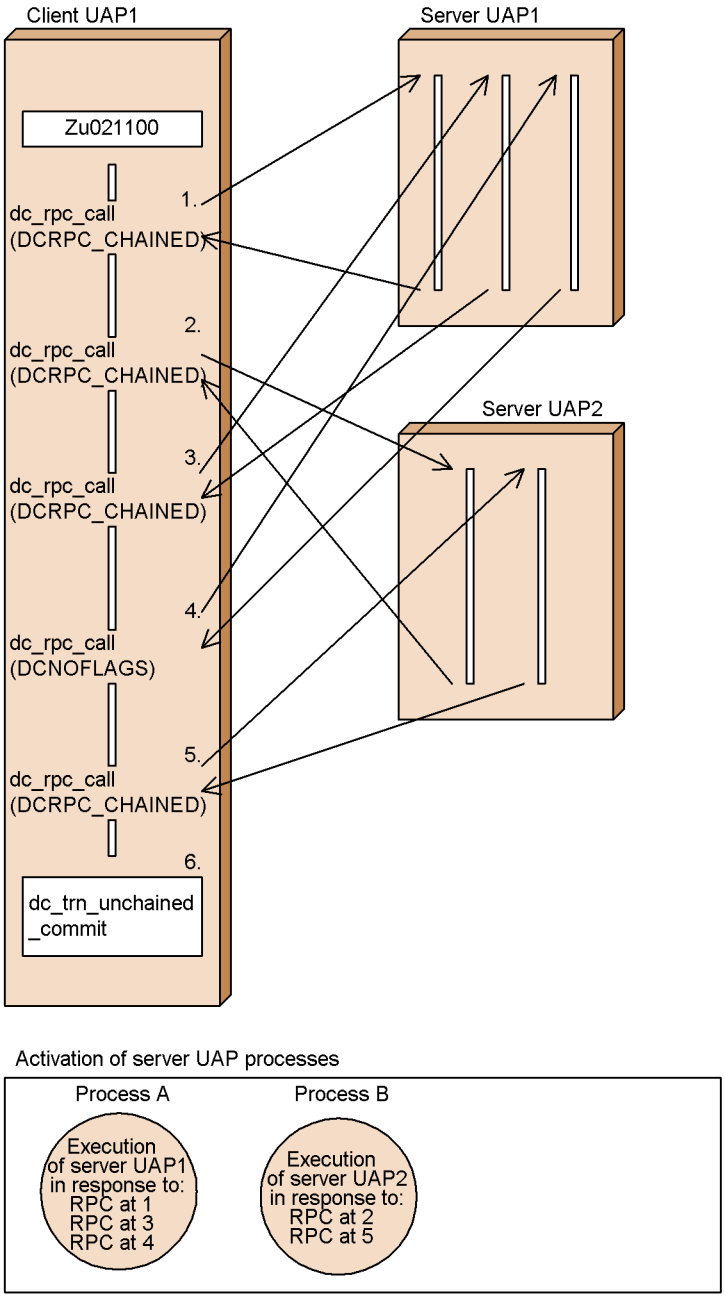
(d) Chained RPCs to servers that receive requests from socket

Servers that receive requests from socket (SPPs which receive service requests without using a schedule queue) do not work as multiserver. They do not work as nonresident processes, either. They work as one server with resident process.

When a service request carried by a chained RPC is executed to a server that receives requests from socket, the server UAP can receive service requests only from the pertinent client UAP. As far as possible, avoid requesting services to servers that receive requests from socket through chained RPCs.

The figure below shows the relationship between chained RPCs and processes.

Figure 2-11: Relationship between chained RPCs and processes



1. When client UAP1 requests server UAP1 for service using a chained RPC

(DCRPC_CHAINED specified for flags), server UAP1 is run by process A.

2. When client UAP1 requests server UAP2 for service using a chained RPC, server UAP2 is run by process B.
3. When client UAP1 requests server UAP1 for service using a chained RPC again, server UAP1 is run by the same process A as in 1.
4. When client UAP1 requests server UAP1 for service using a synchronous-response-type RPC (DCNOFLAGS specified for flags), server UAP1 is run by process A. The chained RPCs for linkage between client UAP1 and server UAP1 are terminated.
5. When client UAP1 requests server UAP2 for service using a chained RPC, server UAP2 is run by process B.
6. When the synchronization point is acquired, the chained RPCs for linkage between client UAP1 and server UAP2 are terminated.

2.1.12 Notes on using a recursive call

The server UAP being executed can be requested by respecifying the same service group name and service name as specified for the service. This feature is called a *recursive call*. When a recursive call is used, a new process is necessary for executing the same service. Therefore, there might be no more processes to be executed upon service request specification when a recursive call is used. In this case, an RPC timeout occurs or the permanent wait state is placed if a wait time is not specified. Specify a sufficient number of processes when using a recursive call. Only queue-receiving servers can use recursive calls. Servers that receive requests from a socket cannot use recursive calls.

A recursive call can also be used in a transaction branch which is a component of the global transaction. However, even if the requested service belongs to the same service group as for the client UAP, the requested service is executed as another transaction branch in another process.

(1) Relationship between a recursive call and the system definition

Even if a recursive call is used, the number of processes does not increase depending on the value set in balance_count (service request remain value) of the user service definition. Consequently, a timeout occurs. Specify 0 as the balance_count value in the following cases:

- A recursive call is used with a user server comprising only nonresident processes (e.g., when parallel_count = 0 or 2).
- A recursive call is used with a server comprising one resident process and nonresident processes (e.g., when parallel_count = 1 or 2).

The following services cannot use a recursive call:

Services belonging to the service group for which `1` is specified as the maximum number of processes in the user service definition (`parallel_count = 1`)

2.1.13 Retrying a service function

If a problem (such as a deadlock) from which the system can be recovered by a retry occurs, you can retry the server UAP process without returning an error to the client UAP. This function is useful when you want to retry a service function in order to eliminate the necessity of reissuing a service request from the client UAP.

To retry a service function, invoke the function `dc_rpc_service_retry()` [`CBLDCRPC('SVRETRY ')`] from within the service function. Then, when the service function is made to return, it is restarted in the same process.

When a service function is retried, the settings made so far by the service function (response storage area and response length) are invalidated.

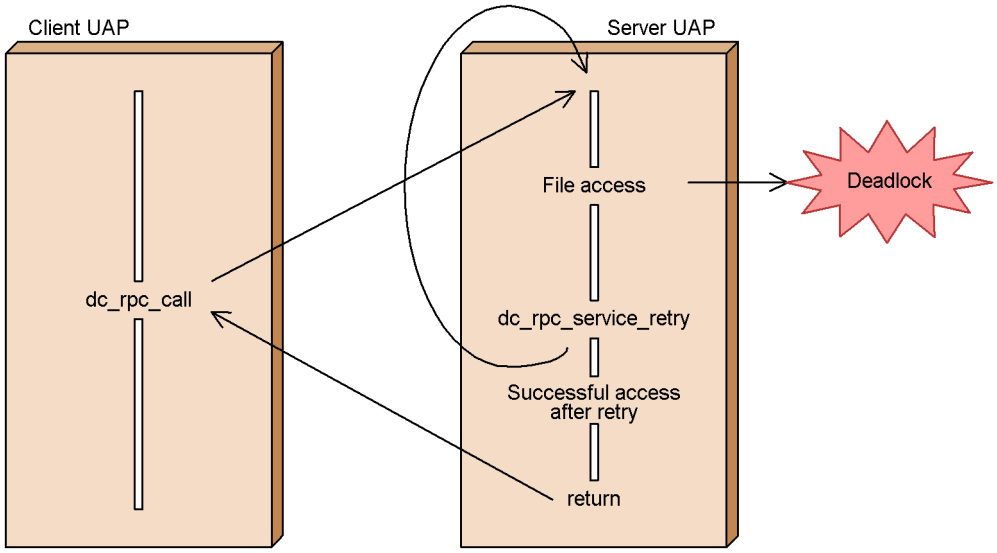
How many times to retry the service function should be assigned to the `rpc_service_retry_count` operand in the user service definition. If the number of retries exceeds the value assigned to this operand, the function `dc_rpc_service_retry()` returns with an error. Service functions that return after this event will not be retried, but will return the value set in the response area to the client UAP.

Before the function `dc_rpc_service_retry()` can be used, the following conditions must be fulfilled. Otherwise, the function returns with an error.

- The function `dc_rpc_service_retry()` is invoked from within the service function.
- The running service function is not within the range of a global transaction.

The figure below outlines the retry of a service function.

Figure 2-12: Outline of retry of service function



2.1.14 User data compression

The user data to be exchanged through RPCs can be compressed so that the number of packets transmitted on the network is reduced and the load on the network is decreased. To compress the user data, specify `Y` in the `rpc_datacomp` operand of the system common definition of OpenTP1 on the client side.

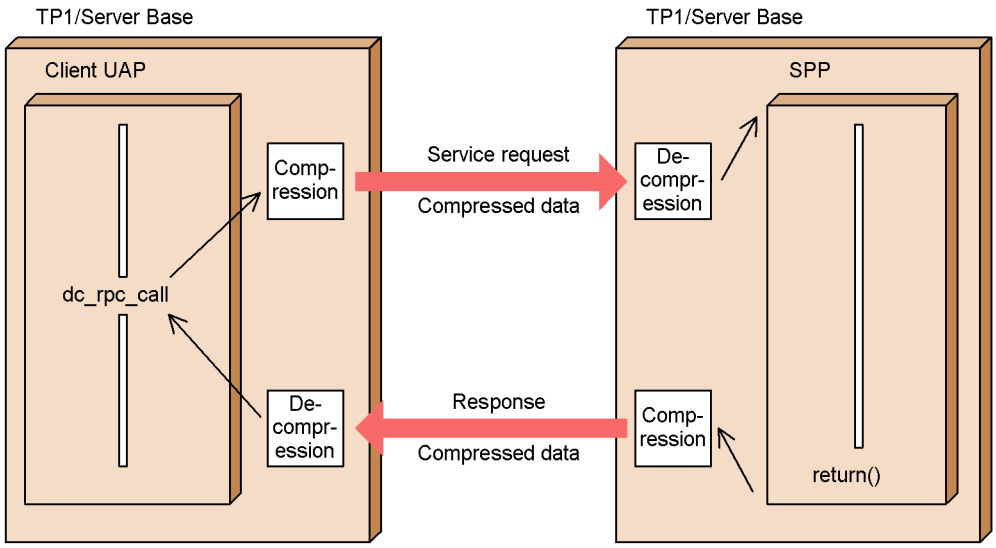
(1) Data compression facility

With the data compression facility, the client side OpenTP1 transmits the service request from the client UAP on the network after compressing the service request data. The corresponding response returned from the SPP is also transmitted on the network after the response data is compressed by the server side OpenTP1. When the client side OpenTP1 receives the response, it decompresses the compressed data and passes it to the client UAP.

The `rpc_datacomp` operand specified is made effective on the client side that requests the service through the function `dc_rpc_call()`. That is, when the `rpc_datacomp=Y` is specified in the client side OpenTP1, the service request and response messages can be transmitted on the network with user data compression even if it is not specified in the server side OpenTP1. Conversely, without specifying `rpc_datacomp=Y` in the client side OpenTP1, the user data for the service request and response messages is not compressed even if it is specified in the server side OpenTP1. This is true only when the server side OpenTP1 supports the user data compression facility.

The figure below shows the outline of the data compression facility.

Figure 2-13: Outline of the data compression facility



(2) Effect of the data compression facility

The effect of the data compression facility depends on the description of the user data. The data compression facility has effect on the user data containing many strings that are composed of repetitive same characters. However, it has no effect on the user data without repetitive same characters.

For the service request whose user data is not affected by compression, it is sent without data compression even if `rpc_datacomp=Y` is specified in the client side OpenTP1. However, if the user data for the corresponding response message is positively affected by compression, the response message is sent after data compression. This is true only when 03-06 or higher version of TP1/Server Base is installed on both the client and server sides. In the other versions, the response message is sent without data compression when the service request data is not compressed.

The data compression facility introduces overhead in data compression and decompression. To use the data compression facility, pre-evaluate its effects and impact on the performance.

2.1.15 Monitoring the service function execution time

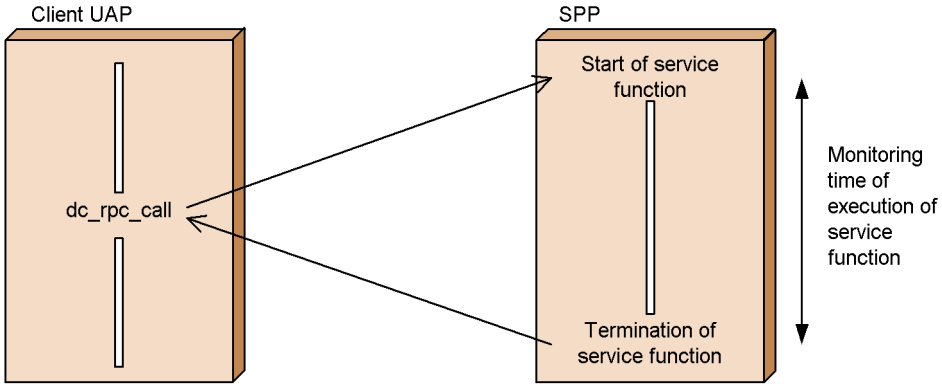
The execution time from when an SPP service function is started until it is terminated can be monitored. When the dynamic loop occurs in the user-created service function because of an error, this facility can be used to cancel such a service. When the service function does not return after the designated monitoring time has elapsed, this facility forces this SPP process to terminate.

To monitor the execution time of the service function, specify a value of `service_expiration_time` in the user service definition.

This facility can only be used in an SPP, not in an MHP. This facility cannot be used in an SPP that operates with the XATMI interface using the OSI TP protocol or with TxRPC interface using the DCE protocol.

The figure below shows the outline of the service function execution time monitoring.

Figure 2-14: Outline of service function execution time monitoring



2.1.16 RPC with the multi-scheduler facility

A client UAP can use a combination of the following three types of RPC on a single OpenTP1 system by using the multi-scheduler facility to request a service provided by a queue-receiving server (SPP that uses the schedule queue) on a remote node.

1. Ordinary RPC

   This method randomly selects one of the OpenTP1 systems containing a service request destination server and sends the service request to the master scheduler daemon of that OpenTP1 system.

2. RPC specifying multiple ports

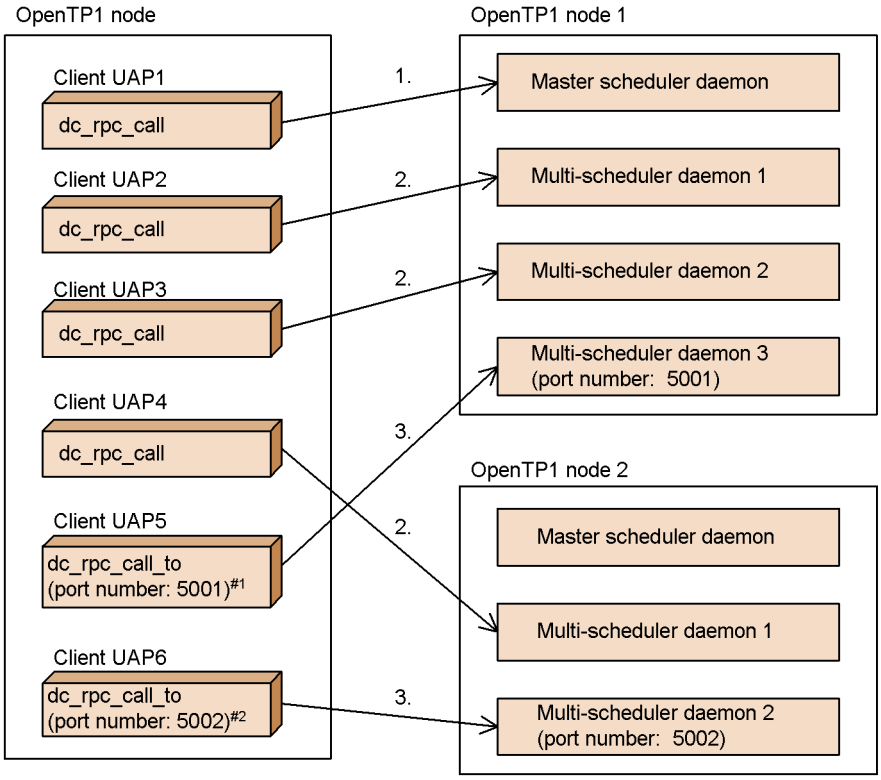
   This method randomly selects one of the multi-scheduler daemons of all OpenTP1 systems containing service request destination servers and sends the service request to that multi-scheduler daemon.

3. RPC with a communication destination specified

   This method sends the service request to the multi-scheduler daemon having the port number specified in the arguments of the function `dc_rpc_call_to()`.

The figure below shows an outline of an RPC that uses the multi-scheduler facility.

Figure 2-15: Outline of an RPC with the multi-scheduler facility



1. Ordinary RPC (not using the multi-scheduler facility)
2. RPC specifying multiple ports
3. RPC with a communication destination specified

#1 Specify the port number (5001) of multi-scheduler daemon 3 on OpenTP1 node 1 in the arguments of the function dc_rpc_call_to().

#2 Specify the port number (5002) of multi-scheduler daemon 2 on OpenTP1 node 2 in the arguments of the function dc_rpc_call_to().

A service request that uses the multi-scheduler facility is scheduled only to nodes on which the multi-scheduler daemon is active. However, if there is no available OpenTP1 system on which the specified multi-scheduler daemon is active at the time the service request is issued, the service request is sent to the master scheduler daemon.

Specifying the port number of a multi-scheduler daemon when issuing a service request with a specified port number causes the RPC to schedule the service request via the specified multi-scheduler daemon.

If a service request's destination user server is blocked or terminated at the time the

multi-scheduler daemon receives the service request, the service request is sent to a multi-scheduler daemon on another node that uses the multi-scheduler facility. If the specified multi-scheduler daemon does not exist on any other node, the service request is sent to the master scheduler daemon.

If a multi-scheduler daemon terminates abnormally while the system is online, the client UAP allocates an appropriate port number to the multi-scheduler daemon that terminated abnormally and restarts it. The OpenTP1 system does not go down in this case. However, if restart fails twice, the OpenTP1 system goes down.

2.1.17 RPC with a communication destination specified

When you use the function `dc_rpc_call()` to request a service, the client UAP need not be aware of the location of the requested service. This is because the name service provided by OpenTP1 manages this information.

In contrast, by using the function `dc_rpc_call_to()`, you can request a service from a specific service request destination. You cannot specify a domain qualifier in the function `dc_rpc_call_to()` for requesting a service. In all other respects, the function `dc_rpc_call_to()` is the same as the function `dc_rpc_call()`.

TP1/Extension 1 must be installed before you can use this function. Note that operation will be unpredictable if you run the function while TP1/Extension 1 is not installed. You can use only a UAP created in C under the control of TP1/Server to call this function. You cannot call it using a UAP created in COBOL.

To designate a service request destination, you must specify one of the following in the arguments of the function `dc_rpc_call_to()`.

1. Host name

   Specify a host name that can be mapped to an IP address with the `/etc/hosts` file or DNS to designate the request destination node.

   In this case, the value specified in the `name_port` operand of the system common definition on the service request destination and the value specified in the `name_port` operand on the service request source (the side that called the function `dc_rpc_call_to()`) must be the same.

2. Node identifier

   Specify the node identifier specified in the `node_id` operand of the system common definition to designate the destination OpenTP1 node for the service request.

   The host name of the destination OpenTP1 node corresponding to the specified node identifier must exist within the global domain.#

3. Host name and port number

   Specify the service request destination by specifying one of the following values.

- Host name that can be mapped to an IP address with the `/etc/hosts` file or DNS.
- Port number of the name service specified in the `name_port` operand of the system common definition of the OpenTP1 system on the host specified above.

In this case, the value specified in the `name_port` operand of the service request destination and the value specified in the `name_port` operand of the service request source need not be the same.

#

Here, *global domain* refers to one of the following sets of node names:

When `N` is specified for the `name_domain_file_use` operand of the system common definition:

A set of node names specified by the `all_node` and `all_node_ex` operands of the system common definition

When `Y` is specified for the `name_domain_file_use` operand of the system common definition:

A set of node names specified in the domain definition file. Note that the domain definition file is stored at the following location:

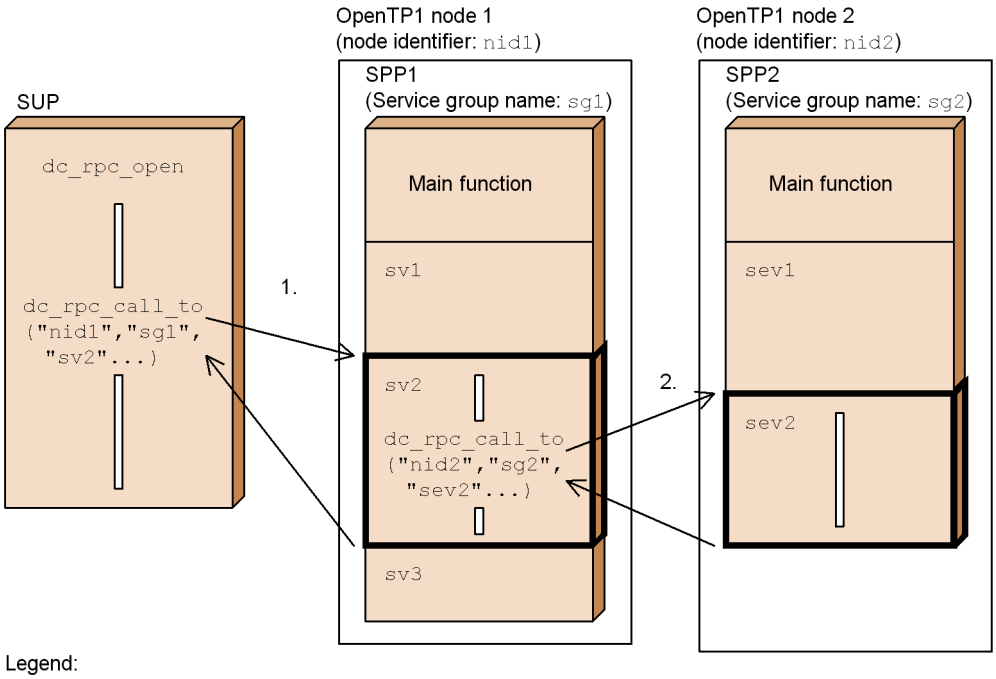
- `all_node` domain definition file

  Under the `$DCCONFPATH/dcnamnd` directory
- `all_node_ex` domain definition file

  Under the `$DCCONFPATH/dcnamndex` directory

The figure below shows an example of communication using the function `dc_rpc_call_to()` in which a node identifier is specified for designating the service request destination.

Figure 2-16: Example of communication using the function dc_rpc_call_to()



Legend:
1. Requests a service that has node identifier nid1, service group name sg1, and service name sv2 from the client UAP.
2. From service sv2 of SPP1, which received a service request, the service that has node identifier nid2, service group name sg2, and service name sev2 can be requested.

2.1.18 Service request with domain qualification

When a service is requested, OpenTP1 searches the entire network constituting the system for the communication partner. Therefore, as the network becomes larger, scheduling the service request takes more time. In order to resolve this problem, the network can be divided into DNS domains for requesting a service. When a service is requested within a domain, OpenTP1 searches the domain for the partner, and the performance on scheduling is improved.

Specify the service group name, an argument of the function dc_rpc_call(), suffixed by the DNS domain name for domain qualification. For service requests with domain qualification, see the description on the function dc_rpc_call() in the applicable *OpenTP1 Programming Reference* manual.

(1) Prerequisites for requesting a service with domain qualification

Prerequisites for requesting a service with domain qualification are as follows:

1. The name of the host on which the domain-alternate schedule service is to be

activated must be registered to the DNS with the `namdomainsetup` command.

2. The port number of the domain-alternate schedule service must be specified in the `scd_port` operand of the schedule service definition of the OpenTP1 which activates the domain-alternate schedule service.
3. The port number of the domain-alternate schedule service specified above must be registered in `/etc/services` of the host on which OpenTP1 that requests a service with domain qualification is to be activated.

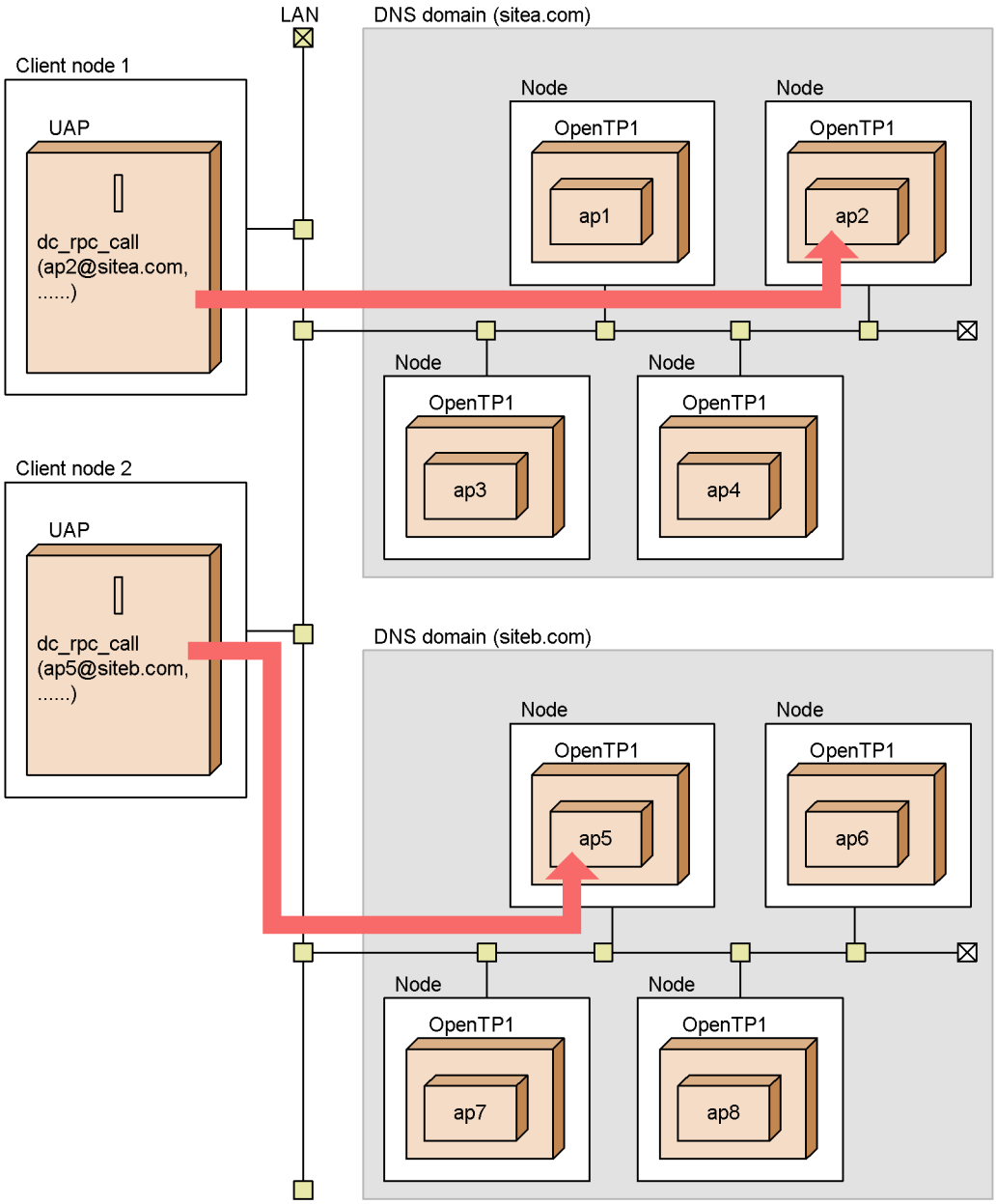
(2) Restrictions on requesting a service with domain qualification

Restrictions on requesting a service with domain qualification are as follows:

1. A service request with domain qualification can be addressed only to a queue-receiving server, rather than a server that receives requests from socket.
2. Even if a service is requested from a transaction, the requested service processing is not treated as a transaction branch.

The figure below shows the outline of service request with domain qualification.

Figure 2-17: Outline of service request with domain qualification



2.1.19 Relationship between service functions and stubs

There are two ways to create service functions in an SPP or MHP.

1. By using a stub
2. By using dynamic loading of service functions

These methods are described below.

(1) Using a stub

Stubs are required to communicate between UAPs using RPCs. A stub is a program which corresponds the service group name and service name specified by the client UAP to the server UAP service.

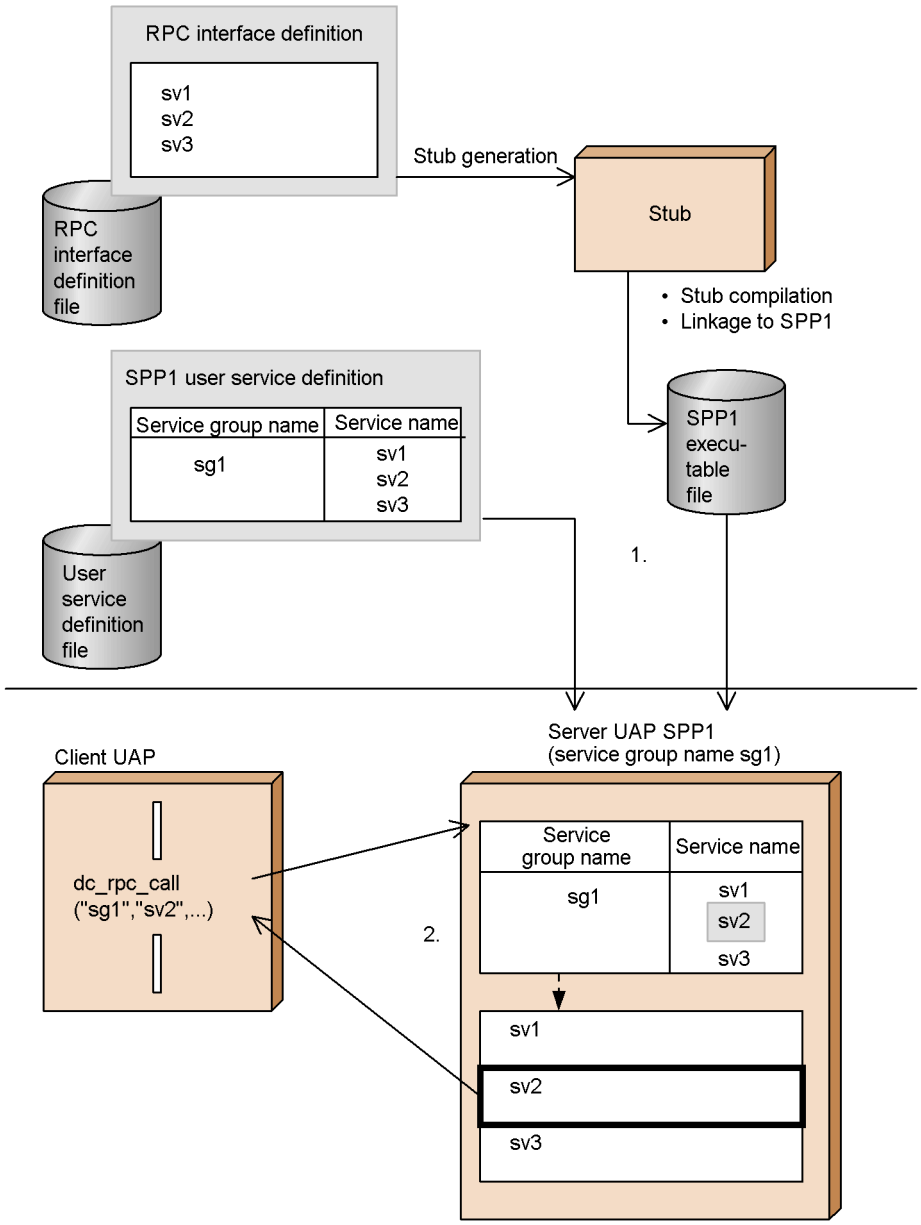
The stub defines the point of entry (entry point) for each UAP service.

When creating a server UAP, link the stub to the object file of the server UAP.

For SUP and UAP that handles offline work, there is no need to define and link the stub.

The figures below show how service functions are created using a stub, for SPP and for MHP separately.

Figure 2-18: Using a stub to acquire service functions (SPP)



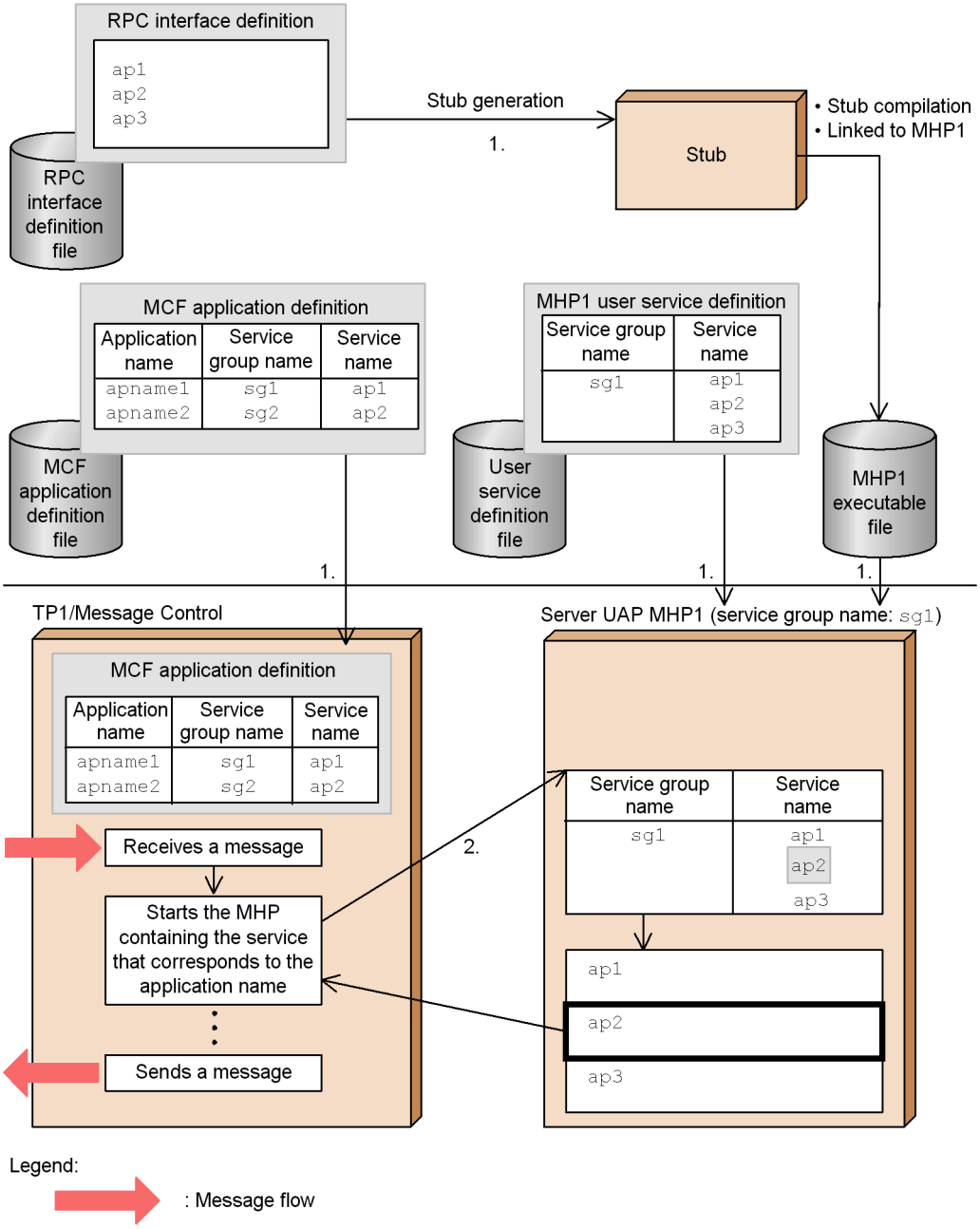
1. The entry point of the service function is defined in the RPC interface definition, and a stub is generated by the `stbmake` command.

   The service group name and the service name are defined in the user service

definition.

2. In the execution form file of the server UAP to which a service was requested, the library created according to the stub and the user service definition is searched for the corresponding service during execution. The results of service processing are then returned to the client UAP.

Figure 2-19: Using a stub to acquire service functions (MHP)



1. The entry point of the service function is defined in the RPC interface definition,

and a stub is generated by the stub-generating command.

The application name, service group name, and service name are associated with each other in the MCF application definition. The service group name and service name are defined in the user service definition.

2. During execution, TP1/Message Control searches for the service name that corresponds to the application name based on the MCF application definition, and starts the corresponding server UAP. In the execution form file of the server UAP to which a service was requested, the library created according to the stub and user service definition is searched for the corresponding service. The service is then processed and service completion is communicated to TP1/Message Control.

(2) Using dynamic loading of service functions

When the facility for dynamic loading of service functions is used, the service functions are acquired from a UAP library that specifies the point of entry (entry point) for each UAP service. There is no need to create a stub. Instead, you need to create a UAP shared library# that incorporates the service functions. You can then acquire service functions from the UAP shared library, eliminating the need to incorporate multiple services into the main function.

#

A UAP shared library is created by compiling UAP source files to produce UAP object files, which are then linked to create a shared library.

The figures below show how service functions are created using dynamic loading of service functions, separately for SPP and MHP.

Figure 2-20: Using dynamic loading of service functions only (SPP)

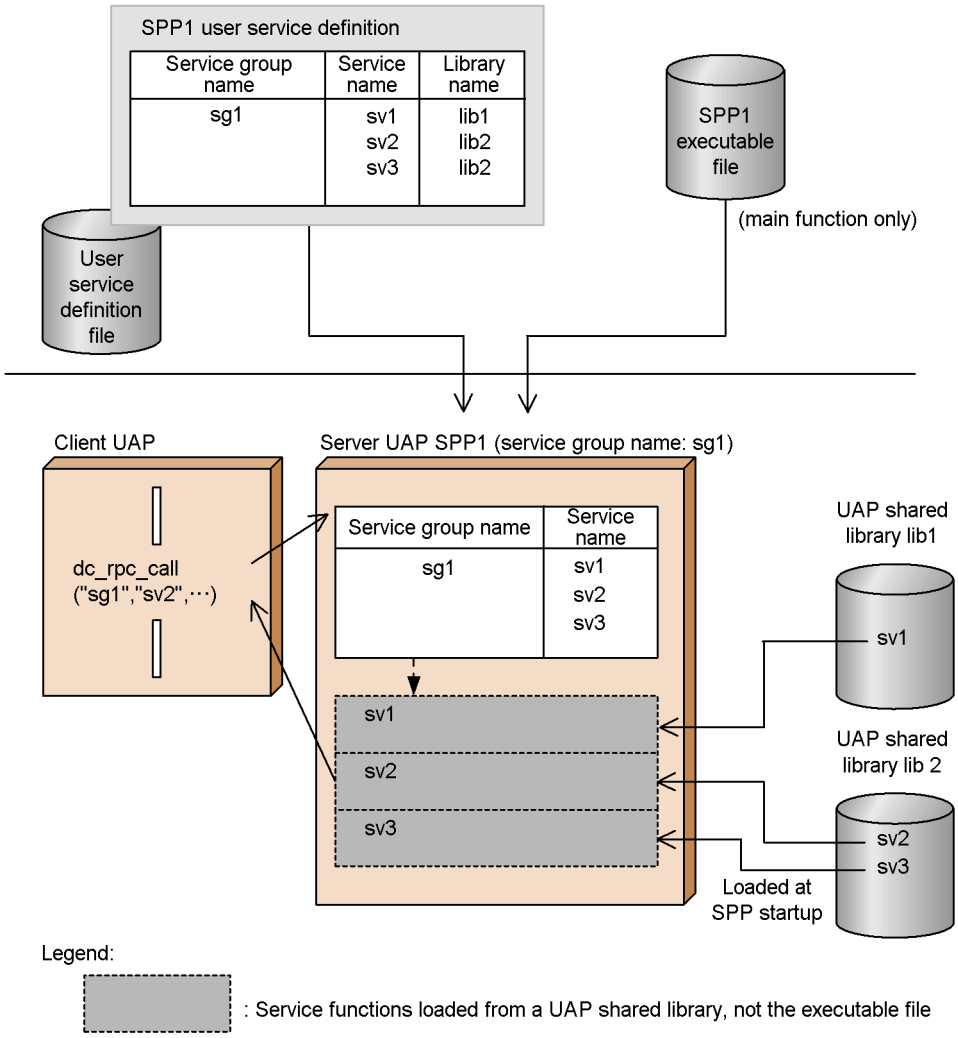
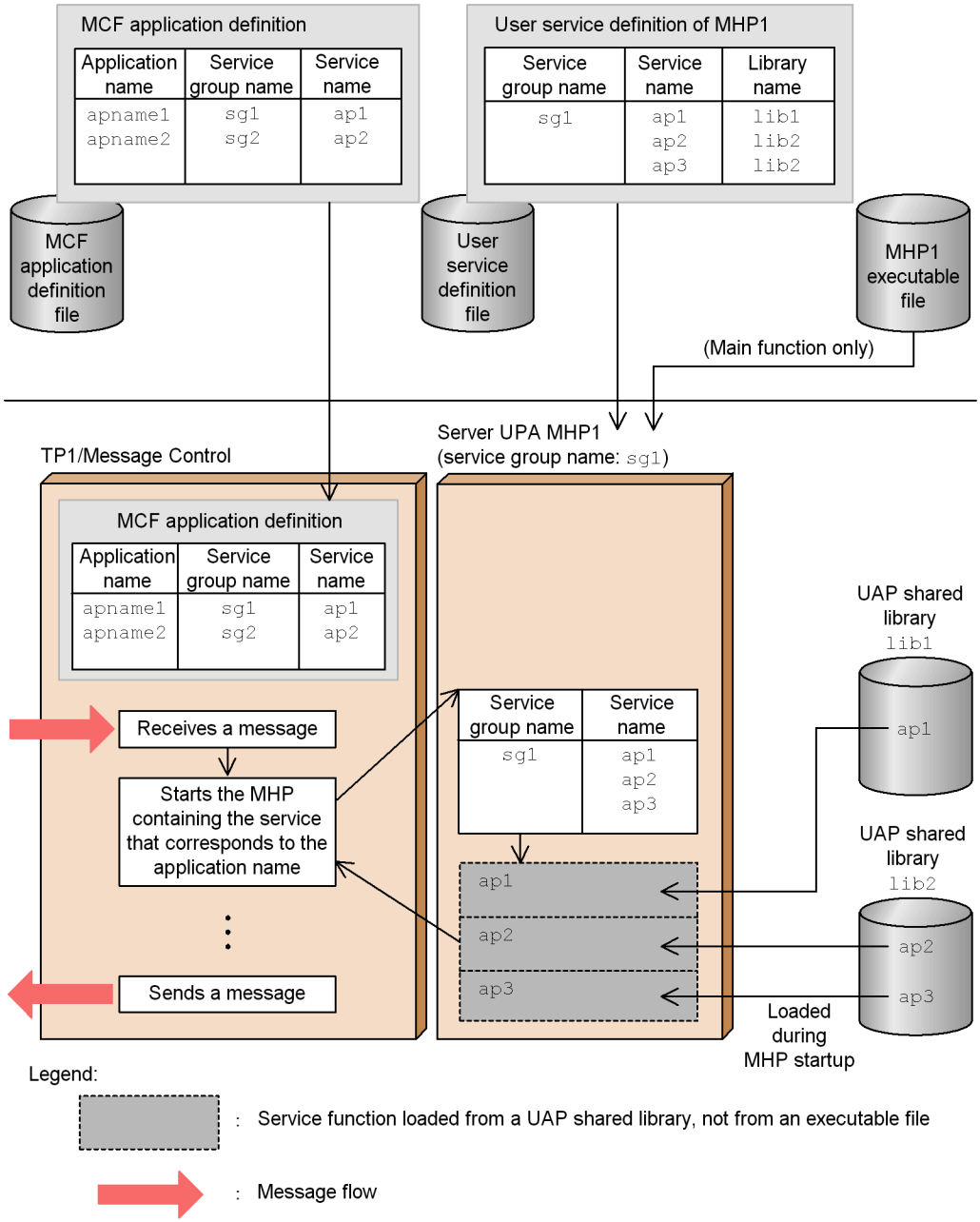


Figure 2-21: Using dynamic loading of service functions only (MHP)



Note that a UAP that uses a stub can also use dynamic loading of service functions. In

this case, a service function can be added without modifying the UAP that uses a stub.

The figures below show, separately for SPP and MHP, how service functions are created using both a stub and dynamic loading of service functions.

Figure 2-22: Using both dynamic loading of service functions and a stub (SPP)

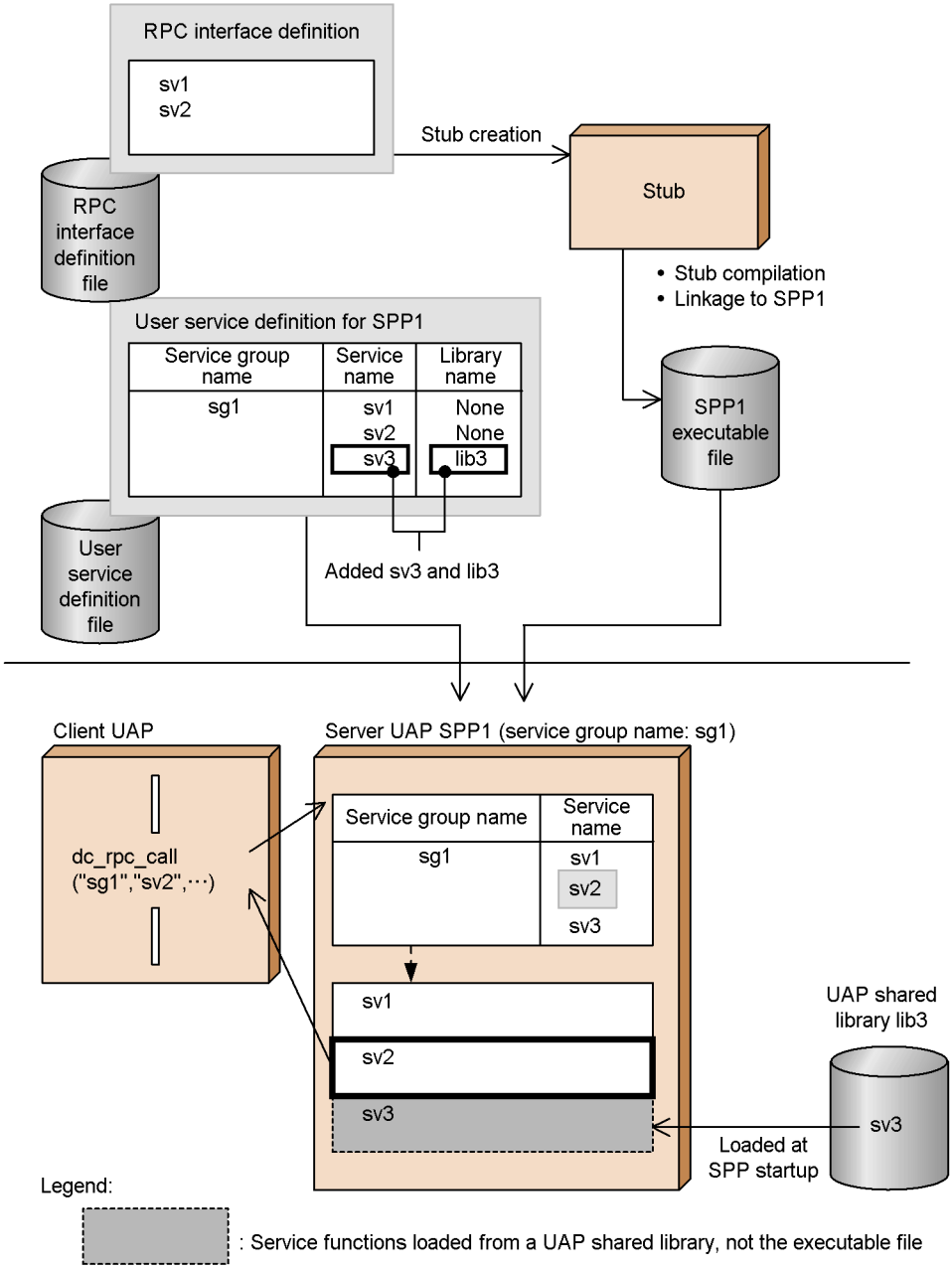
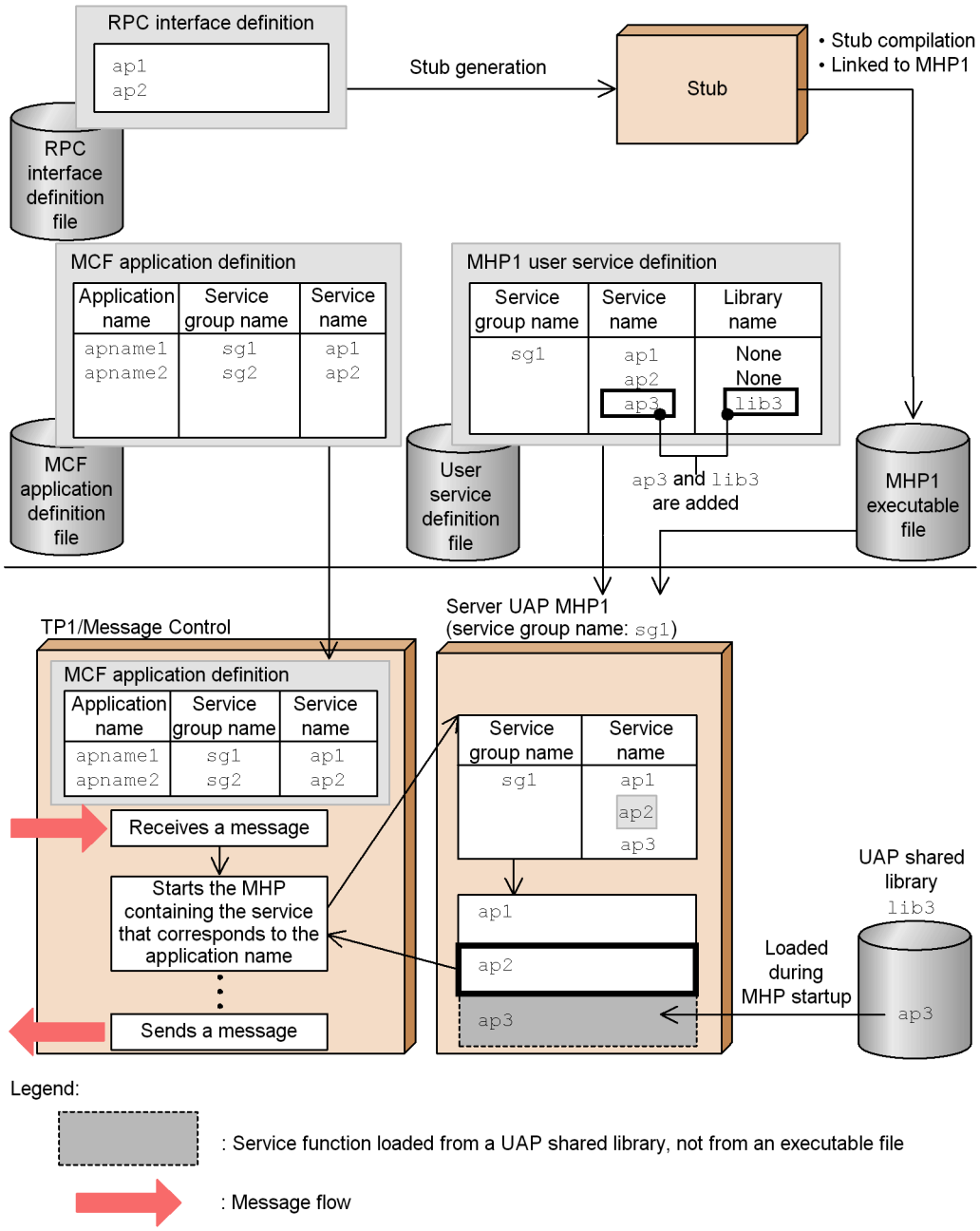


Figure 2-23: Using both dynamic loading of service functions and a stub (MHP)

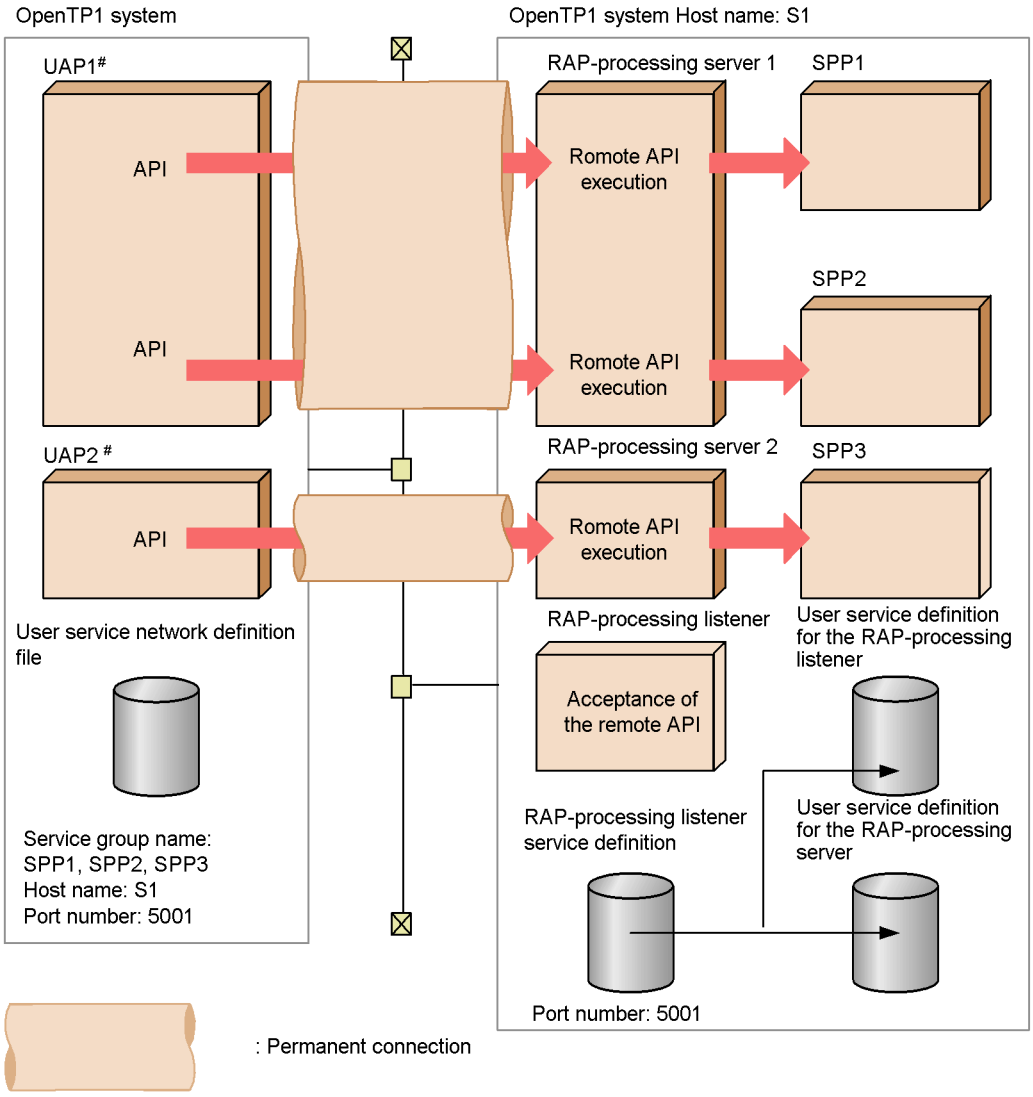


2.2 Remote API facility

When a UAP on a client node issues an API, OpenTP1 can transfer the API to the server for processing on the server. This facility is called the *remote API facility*. A UAP which requests the remote API facility from a client node is called a *RAP-processing client*. The API issued by the RAP-processing client is accepted by the RAP-processing listener on OpenTP1 and is run on the server node by the RAP-processing server. The RAP-processing listener and RAP-processing server run as user services on OpenTP1. You must set up the operating environment for the RAP-processing listener and RAP-processing server using the `rapsetup` command.

To use the remote API facility, define the service information (host name and port number) of the communication destination in the user service network definition. Include the `-w` option in the definition. Create a RAP-processing listener service definition on the server. Also automatically generate user service definitions for the RAP-processing listener and RAP-processing server using the `rapdfgen` command. The figure below shows the remote API facility.

Figure 2-24: Remote API facility



Note The XATMI interface must not be used with the remote API facility.
Operation of OpenTP1 is unpredictable if the XATMI interface is used.

Specify definition in the rpc_destination_mode operand of the user service definition for UAPs on which the remote API facility is to be used.

The following tables show the APIs for which remote execution is possible using the remote API facility for each type of RAP-processing client.

- For RAP-processing clients based on TP1/Server Base or TP1/LiNK

C language library	COBOL-UAP creation program
dc_rpc_call	CBLDCRPC('CALL ')

See the *TP1/LiNK User's Guide* when using TP1/LiNK.

- For RAP-processing clients based on TP1/Client/P or TP1/Client/W

C language library	COBOL-UAP creation program
dc_rpc_call_s	CBLDCRPS('CALL ')
dc_trn_begin_s	CBLDCTRS('BEGIN ')
dc_trn_chained_commit_s	CBLDCTRS('C-COMMIT')
dc_trn_chained_rollback_s	CBLDCTRS('C-ROLL ')
dc_trn_unchained_commit_s	CBLDCTRS('U-COMMIT')
dc_trn_unchained_rollback_s	CBLDCTRS('U-ROLL ')

See the manual *OpenTP1 TP1/Client User's Guide TP1/Client/W, TP1/Client/P* when using TP1/Client/P or TP1/Client/W.

- For RAP-processing clients based on TP1/Client/J

Method
rpcCall
trnBegin
trnChainedCommit
trnChainedRollback
TrnUnchainedCommit
trnUnchainedRollback

See the manual *OpenTP1 TP1/Client User's Guide TP1/Client/J* when using TP1/Client/J.

- For RAP-processing clients based on TP1/Client for .NET Framework

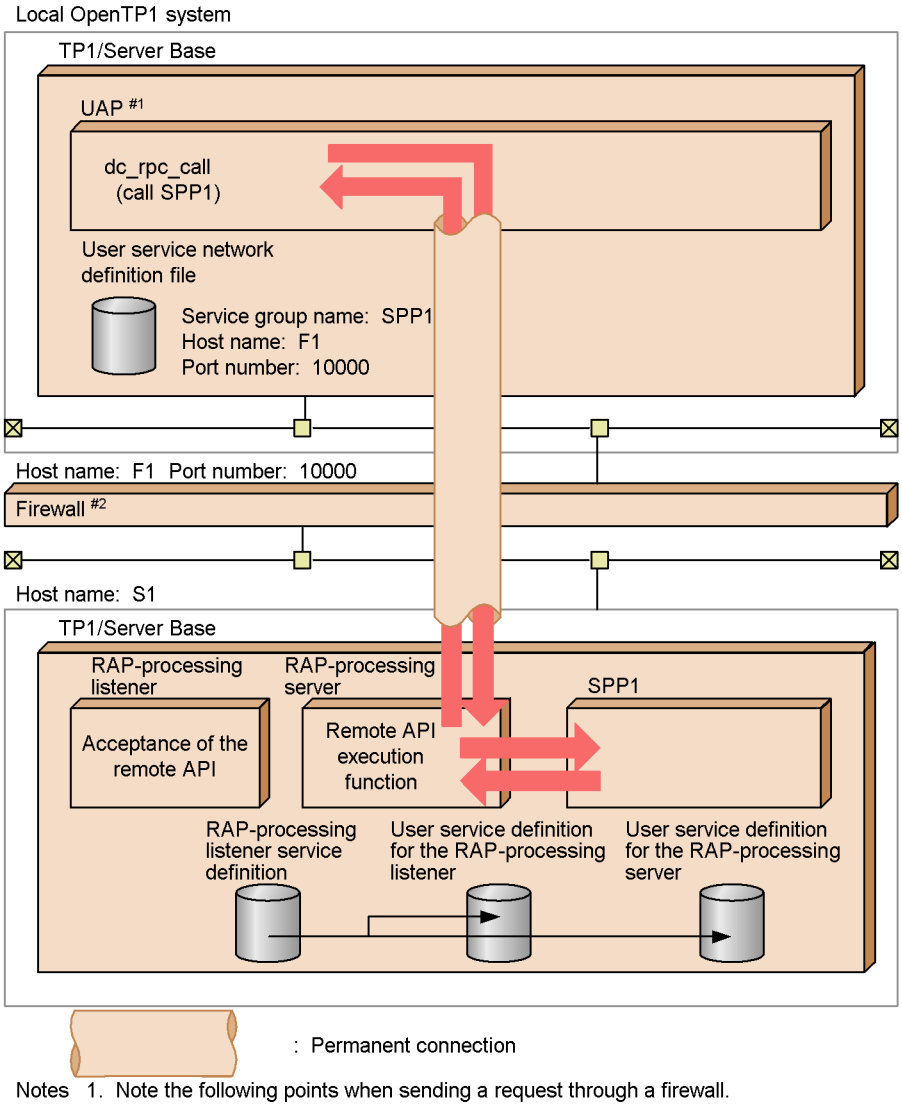
Method
Call
Begin

Method
CommitChained
RollbackChained
Commit
Rollback

2.2.1 Application of the remote API facility

You can use the remote API facility to send a service request to a UAP which is within a firewall. The figure below shows the procedure for sending an RPC through a firewall.

Figure 2-25: Remote procedure call to a UAP within a firewall



Notes 1. Note the following points when sending a request through a firewall.
- Do not use an asynchronous-response-type RPC.
- When a request has been sent through a firewall, the destination UAP (SPP1 in this figure) does not become a transaction branch.

2. The XATMI interface must not be used with the remote API facility.
Operation of OpenTP1 is unpredictable if the XATMI interface is used.

#1 Specify definition in the rpc_destination_mode operand of the user service definition for the UAP.

#2 This figure assumes that Gauntlet is used as the firewall.

2.2.2 Permanent connection

OpenTP1 provides a logical channel (*permanent connection*) between the UAP (RAP-processing client) that requested the remote API and the RAP-processing server.

There are two methods of scheduling a permanent connection: Static connection mode and dynamic connection mode. Specify which mode is to be used in the `rap_connection_assign_type` operand in the RAP-processing listener service definition.

2.2.3 Connection mode

The method of managing permanent connections can be classified into two modes according to the method in which the connections are established and released. The mode whereby OpenTP1 manages the establishment and release of connections is called the *automatic connection mode*. The mode whereby the user manages the establishment and release of connections is called the *non-automatic connection mode*. In the user service definition for the RAP-processing client, specify whether automatic connection mode or non-automatic connection mode is to be used for managing permanent connections.

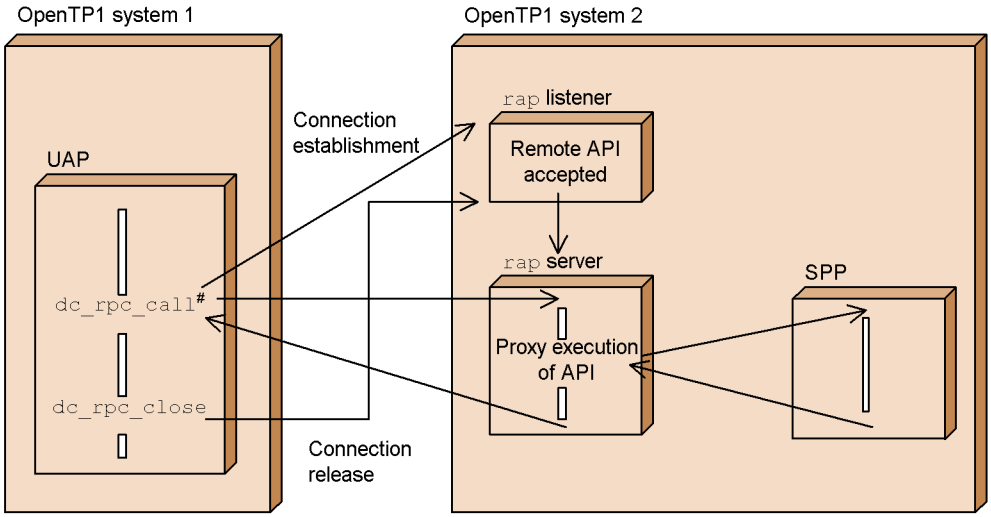
(1) Automatic connection mode

In this mode, OpenTP1 manages the establishment and release of permanent connections. OpenTP1 automatically establishes a permanent connection when a RAP-processing client calls the function `dc_rpc_call()` in which the service group name defined in the user service network definition is specified together with the `-w` option as an argument.

As soon as the RAP-processing client has called the function `dc_rpc_call()` to request a service from the service group defined in the user service network definition, it calls the function `dc_rpc_close()`. It maintains the permanent connection until the RPC has returned.

The figure below shows the outline of the automatic connection mode.

Figure 2-26: Outline of automatic connection mode



Note: OpenTP1 manages the establishment and release of resident connections.
#: The -w option is defined in the user service network definition of the rap client.

There is a limit to the number of connections that can be established between the RAP-processing client and the RAP-processing server. If calling the function dc_rpc_call() causes the number of connections to exceed this limit, OpenTP1 automatically releases the least recently used connection among those used by the RAP-processing client process, and then establishes a new connection.

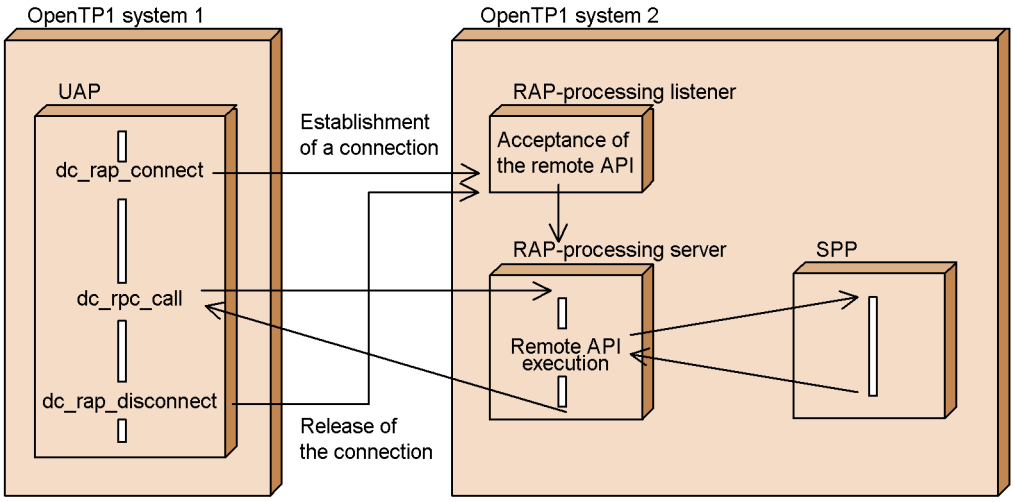
However, OpenTP1 cannot release a connection which is being used by a chained RPC. When OpenTP1 cannot release any connection due to this restriction, the UAP that issued the API goes down.

(2) Non-automatic connection mode

In this mode, the user manages the establishment and release of permanent connections. To establish a connection from a RAP-processing client, call the function dc_rap_connect() [CBLDCRAP('CONNECT')]. To release a connection, call the function dc_rap_disconnect() [CBLDCRAP('DISCNCT')]. The RAP-processing client calls the function dc_rpc_call() in which the service group name defined in the user service network definition is specified together with the -w option as an argument. If the user has not established a permanent connection by the time the RAP-processing client calls the function dc_rpc_call(), the function dc_rpc_call() returns with an error. The return value is DCRPCER_PROTO.

The figure below shows the outline of the non-automatic connection mode.

Figure 2-27: Outline of non-automatic connection mode



If calling the function `dc_rap_connect()` causes the number of connections to exceed the maximum number of connections which the RAP-processing client can establish with the RAP-processing server, the function `dc_rap_connect()` returns with an error.

2.2.4 Chained RPCs using the remote API facility

This subsection explains chained RPCs in automatic connection mode and chained RPCs in non-automatic connection mode, both of which are chained RPCs on a permanent connection.

(1) Chained RPCs in automatic connection mode

There is a limit to the number of connections that can be established between the RAP-processing client and the RAP-processing server. If calling the function `dc_rpc_call()` causes the number of connections to exceed this limit, OpenTP1 automatically releases the least recently used connection among those used by the RAP-processing client process, and then establishes a new connection.

However, OpenTP1 cannot release a connection which is being used by a chained RPC. When OpenTP1 cannot release any connection due to this restriction, the UAP that issued the API goes down.

(2) Chained RPC in non-automatic connection mode

The running of a chained RPC may encounter one of the following events: the calling of the function `dc_rap_disconnect()` and failure or communication error of the UAP that issued the API function. In such a case, the RAP-processing server that is performing remote execution of the API terminates abnormally and restarts. The

purpose of this momentary termination is to reset the connection with the SPP that is processing the chained RPC and clear the status.

2.2.5 Notes on the remote API facility

Note the following points when using the remote API facility:

- Do not use an asynchronous-response-type RPC with the remote API facility. If this type of RPC is used, the remote API facility is ineffective and the RPC works as a normal RPC.
- Do not use the XATMI interface with the remote API facility. Operation of OpenTP1 is unpredictable if the XATMI interface is used.
- If a RAP-processing client uses the remote API facility to call the function `dc_rpc_call()` from within a transaction, the requested service is not run as a transaction.
- You cannot acquire RPC trace information on communication that was performed using the remote API facility. However, information on the function `dc_rpc_call()` that was executed on the RAP-processing server is obtained in the RPC trace information area.
- The response statistical information and communication delay time statistical information do not include information on services and results that were communicated using the remote API facility.
- When executing an RPC via the gateway of an application gateway-type firewall, you might use the remote API facility with the `-w` option specified in the `dcsvgdef` definition command of the user service network definition. In this case, even if you call the function `dc_rpc_call()` with the transaction attribute, it is not regarded as a transaction. Therefore, when you have used the remote API facility, the operation for starting chained RPCs from within a transaction and terminating the chained RPCs by means of synchronization point processing does not execute properly. Terminate the chained RPCs explicitly by calling the function `dc_rpc_call()` with `DCNOFLAGS` specified in `flags`.
- Normally, the RAP-processing server is started automatically by the RAP-processing listener. Do not independently terminate (by executing the `dcsvstop` command) or start (by executing the `dcsvstart` command) the RAP-processing server. However, in the following cases, use the `dcsvstart` command to start the RAP-processing server.

  When the RAP-processing server fails to start due to a definition error:

  Even when you cannot start the RAP-processing server due to a definition error or other problem, the RAP-processing listener cannot detect the failure to start the RAP-processing server. The system therefore remains in the status that indicates preparation of the remote API service (the status when

the message KFCA26950-I is output). If the message KFCA01812-E with the error reason code CONFIGURATION is output when the RAP-processing listener starts the RAP-processing server, check the definition of the RAP-processing server and use the dcsvstart command to start the RAP-processing server. Note that the message KFCA00244-E cannot detect a definition error for the RAP-processing server.

When the RAP-processing listener goes down while the RAP-processing listener and the RAP-processing server are terminating:

After the RAP-processing listener goes down, even if you start the RAP-processing listener using the dcsvstart command, the RAP-processing server outputs the KFCA26950-I message and the system may remain in the status that indicates preparation of the remote API service. If the RAP-processing server is not started, start it by executing the dcsvstart command.

- Do not execute the scdhold command for the RAP-processing server while the RAP-processing server is online.
- Do not call a service request that uses the remote API facility for a UAP on the same node as the RAP-processing server. Processing cannot be guaranteed in such a case.

2.3 Transaction control

OpenTP1 can perform transaction control in client/server mode. This facility enables the UAP processing over multiple processes to be executed as one transaction. There are two functions for transaction control applicable to OpenTP1 UAPs:

- OpenTP1 specific interface
- TX interface (transaction control conforming X/Open specifications)

This section explains OpenTP1 specific interface. For details on TX interface, see *5.2 TX interface (transaction control)*.

This section explains transaction control involved in UAPs (SUP, SPP) in client/server mode. For details on transaction control involved in UAPs (MHP) in message exchange mode, see *3.7 MCF transaction control*.

- Notes on using UAPs with TP1/LiNK

  To implement transaction control through UAPs used with TP1/LiNK, specify that the transaction facility will be used when setting up a TP1/LiNK execution environment.

2.3.1 Transaction in client/server mode

OpenTP1 can implement one transaction with multiple-process RPCs. This transaction is called a *global transaction*. A transaction to be processed in client/server mode can be ensured by implementing this global transaction.

(1) Transaction start and synchronization point acquisition (commitment)

Before transactions can be controlled during client/server mode communication, the transaction start and the acquisition of a synchronization point must be explicitly specified in the UAP.

To start a transaction, invoke the following function:

- `dc_trn_begin()` [`CBLDCTRN('BEGIN   ')`]

To acquire a synchronization point, invoke the following functions:

- `dc_trn_chained_commit()` [`CBLDCTRN('C-COMMIT')`]
- `dc_trn_chained_rollback()` [`CBLDCTRN('C-ROLL  ')`]
- `dc_trn_unchained_commit()` [`CBLDCTRN('U-COMMIT')`]
- `dc_trn_unchained_rollback()` [`CBLDCTRN('U-ROLL  ')`]

The client UAP becomes a root transaction branch when the transaction start function is called. Functions for acquiring transaction synchronization points (commitment) are

called from the root transaction branch from which the transaction start function was called.

After a transaction start function is called, another transaction start function cannot be called in the global transaction.

When a UAP being executed requests a service as a transaction, the service is being executed as a transaction upon the request. The function `dc_trn_begin()` cannot be called with the requested service.

(2) UAPs that can call transaction control functions

Only SUPs and SPPs can call functions which start a transaction or acquire the synchronization point. Since transaction processing for MHPs is automatically controlled by OpenTP1, MHPs cannot call transaction control functions. SPPs which are requested for service by MHPs via the function `dc_rpc_call()` cannot call transaction control functions, either.

UAPs that handle offline work cannot use transaction control functions.

UAPs providing the OpenTP1 client facility (CUPs) use the transaction control functions existing in the TP1/Client library.

2.3.2 Acquiring a synchronization point

The *acquisition of a synchronization point* means to make all transaction branches comprising a global transaction synchronized and terminated with the same result (commitment or rollback).

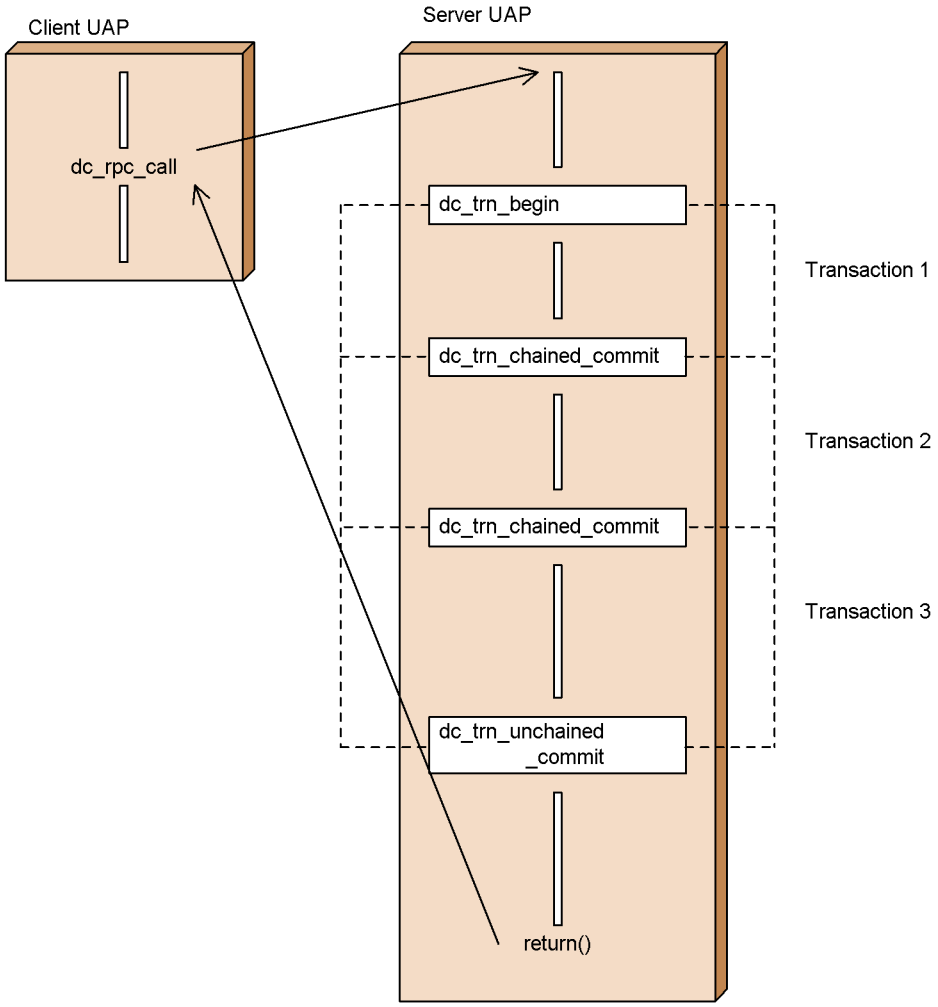
(1) Using commitment functions

Commitment functions can be used only with the SPP or SUP (root transaction branch) that started a transaction using the function `dc_trn_begin()`. Note that commitment functions cannot be used with another transaction branch. The global transaction terminates normally when all transaction branches terminate normally.

(a) Commitment in chained/unchained mode

There are two types of transaction processing commitment: commitment (`dc_trn_chained_commit()`) in chained mode which acquires a synchronization point and consecutively starts the next transaction after one transaction terminates, and commitment (`dc_trn_unchained_commit()`) in unchained mode which does not start a new transaction after one transaction terminates. The figure below shows transactions in chained/unchained mode.

Figure 2-28: Transactions in chained/unchained mode



(2) Using rollback functions

If you want to roll back a transaction according to UAP decision, you can call a rollback request from the UAP.

(a) Rollback in chained/unchained mode

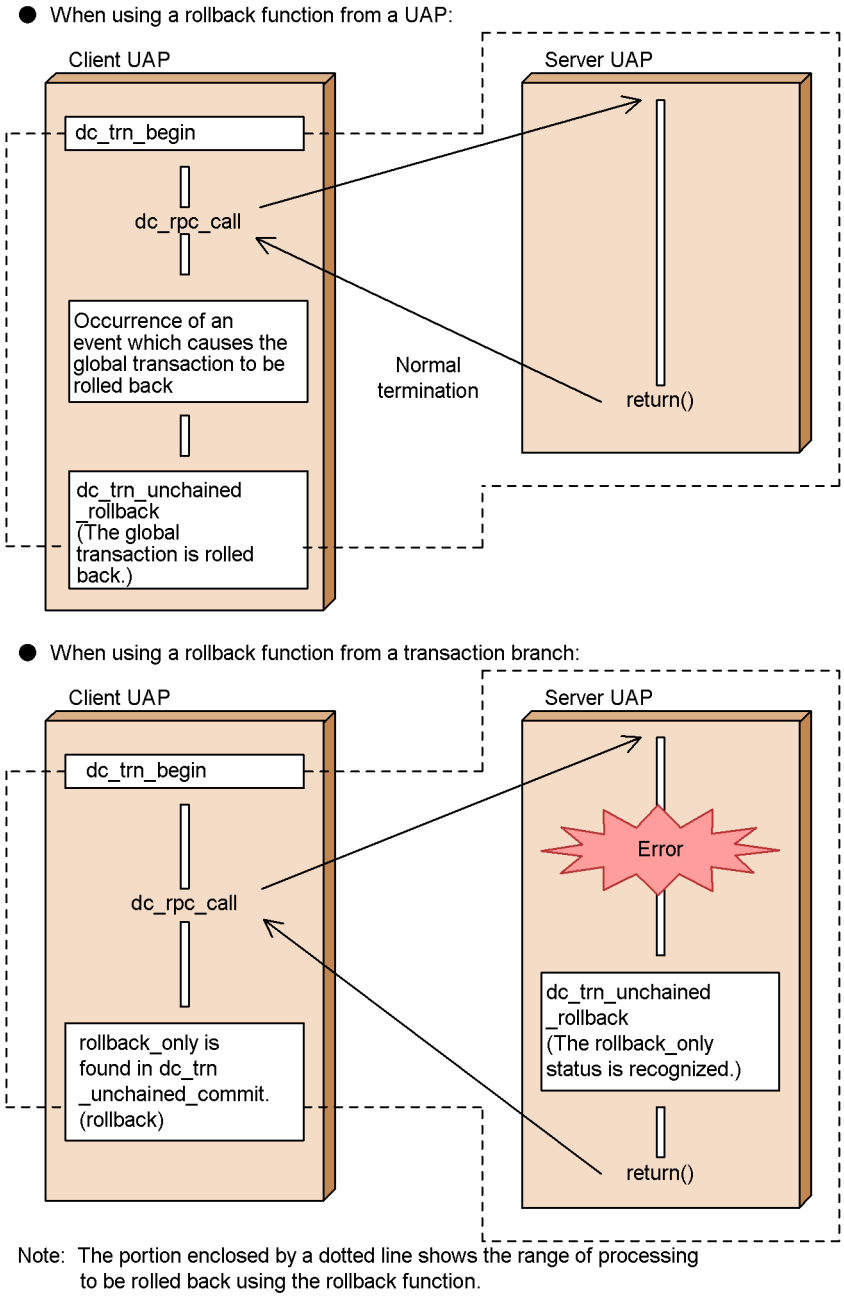
There are two rollback functions: `dc_trn_chained_rollback()` (chained mode) and `dc_trn_unchained_rollback()` (unchained mode). The rollback function of the chained mode remains in the new global transaction range even after rollback processing is executed. If the rollback function of the unchained mode is called from the root transaction branch, the rollback function is not in the global transaction range.

The rollback function of the chained mode cannot be called from the root transaction branch. The rollback function of the unchained mode can be called from any transaction branch.

If the rollback function of the unchained mode is called from a transaction branch, the transaction branch is a rollback target (`rollback_only` status). This information is posted to the root transaction branch. In this case, the rollback function of the unchained mode remains in the global transaction range after rollback processing is executed and before the function `dc_rpc_call()` returns.

The figure below shows transaction rollback.

Figure 2-29: Transaction rollback



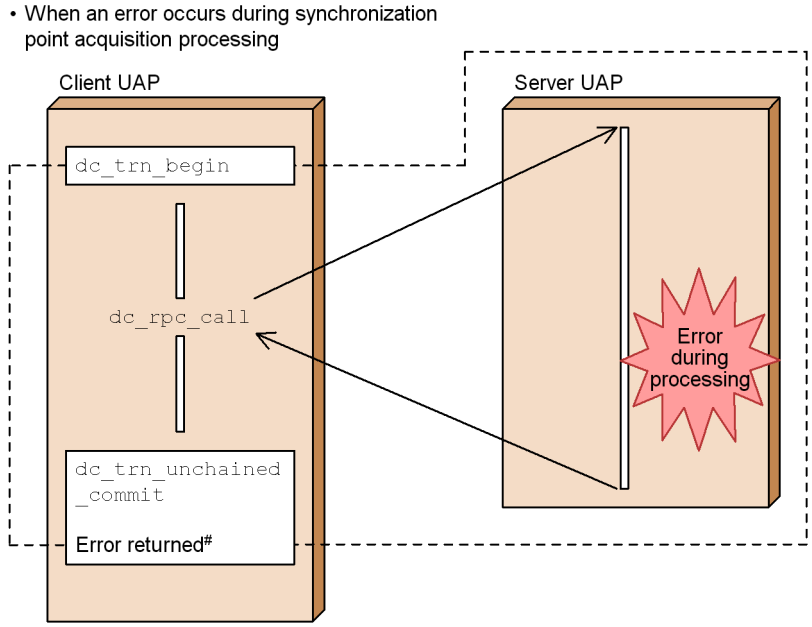
Note: The portion enclosed by a dotted line shows the range of processing to be rolled back using the rollback function.

(3) If an error occurs during synchronization point acquisition processing

If an error occurs during a process for acquiring transaction synchronization points, the transaction is committed when it has been completed to phase 1 of synchronization points; otherwise, it is rolled back. When one of the transaction branches within a global transaction is rolled back, the whole global transaction is rolled back.

The figure below shows transaction rollback in the case of an error occurs during synchronization point acquisition processing.

Figure 2-30: Transaction rollback if an error occurs during synchronization point acquisition processing



#: If the transaction has been completed to Phase 1 of the synchronization point, the processing inside the dotted line is committed. Otherwise, the transaction is rolled back.

(4) Action to be taken if a function for acquiring synchronization point is not called

If a UAP which does not call a function for acquiring a synchronization point terminates abnormally, the result of the UAP synchronization point is rolled back.

If the UAP (root transaction branch) terminates with the function `exit()` without using a function for acquiring a synchronization point, OpenTP1 performs automatic commitment. If this commitment processing encounters an error before it reaches the end of phase 1, the global transaction is rolled back. In this case, this rollback cannot

be posted to the UAP.

2.3.3 Specification of transaction attribute

When setting up a UAP execution environment, specify whether to run UAP processes as transactions. A UAP process is called a *UAP with the transaction attribute* if it is specified so that it will work as a transaction. The transaction attribute must be specified for UAPs which update files or perform other transaction processing.

(1) How to give transaction attribute to UAP

To make a server UAP process a transaction branch, specify that the UAP have the transaction attribute. The transaction attribute is specified by the following method:

- TP1/Server Base:

  Specify Y for `atomic_update` in the user service definition.

- TP1/LiNK:

  Specify for the user server that the transaction facility will be used.

Processing of a UAP with the transaction attribute works as a transaction when:

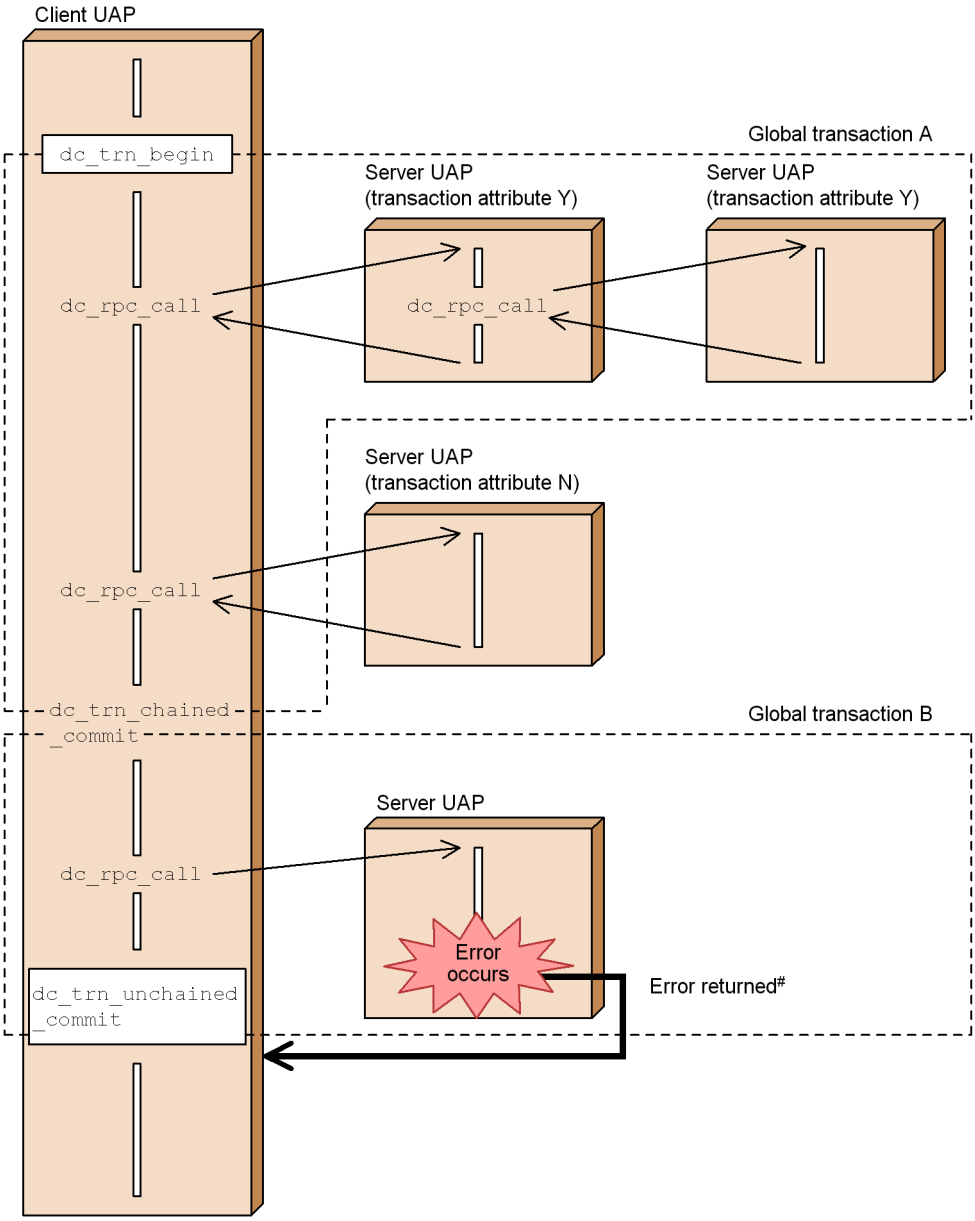
- The UAP with the transaction attribute normally returns by using the function `dc_trn_begin()` to start a transaction.
- The UAP is requested for service via the function `dc_rpc_call()` by another UAP which is working as a transaction.

(2) How to give nontransaction attribute to UAP

The nontransaction attribute (`atomic_update=N` or the specification that the transaction facility be not used) must be specified for server UAPs which perform only operation and other server UAPs which do not require that transactions be guaranteed. Server UAPs with the nontransaction attribute can always offer service to not only the present, but also other client UAPs independent of global transaction processing. Even if the server UAP is requested for service by multiple client UAPs, it can start handling these service requests without waiting until synchronization point acquisition processing is completed. This helps reduce the overhead involved in service request waits.

The figure below shows the relationship between RPCs and the transaction attribute.

Figure 2-31: Relationship between RPCs and transaction attribute



#

The contents of a resource accessed by global transaction B are returned to the status immediately before global transaction B is started. The function for

acquiring a synchronization point (the function `dc_trn_unchained_commit()` in the figure) returns an error to report that global transaction B has rolled back.

(a) Using nontransactional RPC from transaction process

If a transaction process requests a UAP for service and the requested UAP has the transaction attribute, the service request is handled by a transaction process. It is possible to make such service requests not handled by a transaction process. For this purpose, specify the argument to the function `dc_rpc_call()` to indicate that the RPC is nontransactional.

2.3.4 Relationship between remote procedure call modes and synchronization points

If the transaction attribute is specified for an SPP called through an RPC from a UAP (SUP, SPP, MHP) working as a transaction, the SPP works as a transaction. Each transaction branch can be synchronized as one global transaction. Each process of server UAP returns to the UAP that called the function `dc_rpc_call()` after processing terminates. However, the next service request can be accepted only after the service returns to the root transaction branch and synchronization point processing is completed. Resources acquired by the server UAP can also be released after the service returns to the root transaction branch and synchronization point processing is completed. These features also apply when asynchronous-response-type RPCs, nonresponse-type RPCs, or chained RPCs are used.

UAP processing can be synchronized between UAPs through RPC transaction control as explained in the above.

Another service request can be handled in a process of the server UAP before the synchronization point processing has been completed in the client UAP. This is called *transaction optimization*. For details on transaction optimization, see *2.3.5 Transaction optimization*.

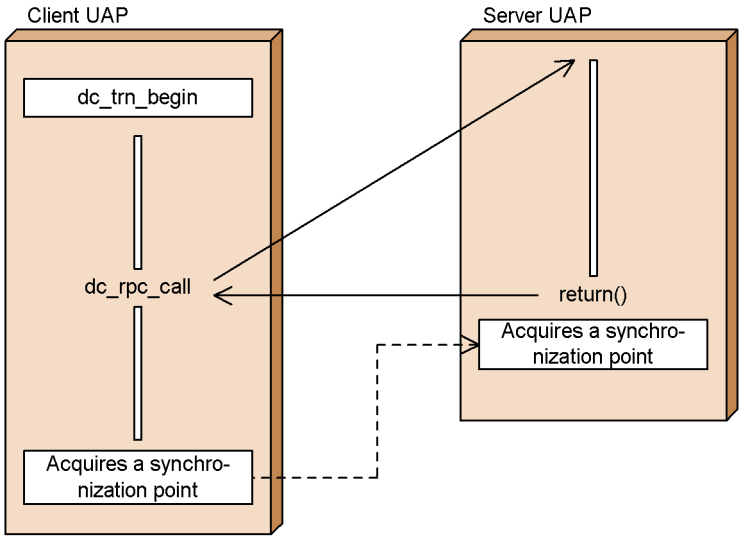
(1) Relationship between synchronous-response-type RPCs and synchronization points

In the case of transaction processing of a synchronous-response-type RPC is executed, the transaction terminates when the processing results are returned to the root transaction branch and synchronization point processing is completed.

If the requirements for optimizing transaction are satisfied, a process of the server UAP can accept the next service request when processing terminates.

The figure below shows the relationship between the synchronous-response-type RPC and the synchronization point.

Figure 2-32: Relationship between synchronous-response-type RPC and synchronization point

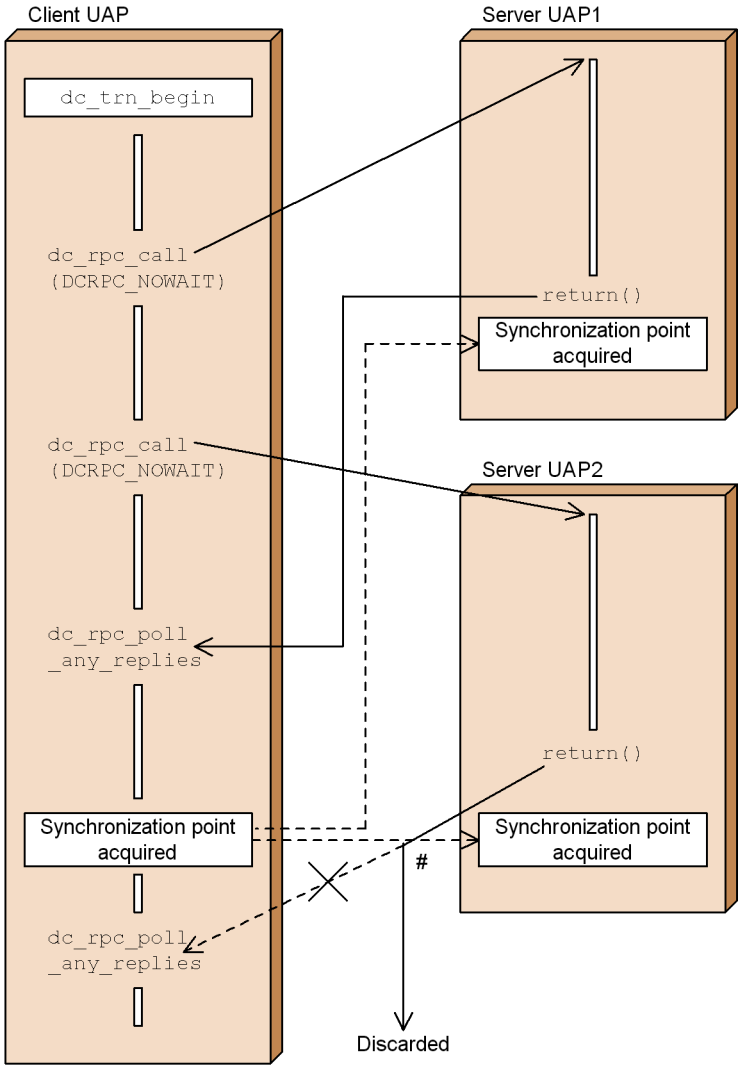


(2) Relationship between asynchronous-response-type RPCs and synchronization points

In the case of transaction processing of an asynchronous-response-type RPC, RPC processing will terminate when the client UAP finishes synchronization point processing. If a response comes from the server UAP after synchronization point processing, the UAP which called the function `dc_rpc_call()` cannot receive the response.

The figure below shows the relationship between the asynchronous-response-type RPC and the synchronization point.

Figure 2-33: Relationship between asynchronous-response-type RPC and synchronization point



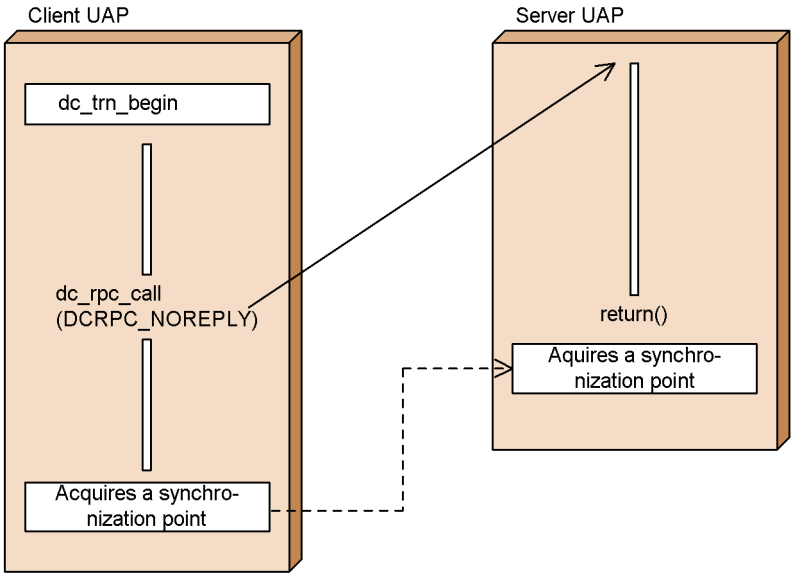
#: Even if the function `dc_rpc_poll_any_replies` is called after synchronization point acquisition by the client UAP is completed, responses cannot be received. To receive all results of services requested by an asynchronous-response type RPC, the function `dc_rpc_poll_any_replies` must be called before synchronization point acquisition for each time the function `dc_rpc_call` is called. To reject responses, the function `dc_rpc_discard_further_replies` must be called.

(3) Relationship between nonresponse-type RPCs and synchronization points

In the case of transaction processing of a nonresponse-type RPC, the client UAP waits until the server UAP finishes processing, and then executes synchronization point acquisition processing.

The figure below shows the relationship between the nonresponse-type RPC and the synchronization point.

Figure 2-34: Relationship between nonresponse-type RPC and synchronization point



(4) Relationship between chained RPCs and synchronization points

Chained RPCs are executed by one server UAP process. Therefore, there is one transaction branch regardless of the number of chained RPCs used.

In the case of transaction processing of chained RPCs, the transaction will terminate when synchronization point processing is terminated. The server UAP process is then freed.

If non-transactional chained RPCs are used during the transaction, the server UAP process in charge of the processing will, in general, be freed when the synchronous point processing ends. If you want to free the server UAP process in charge of the processing by means of a synchronous-response-type RPC (with `DCNOFLAGS` assigned to `flags`) rather than the termination of the synchronous point processing, assign `00000002` to the user service definition `rpc_extend_function` operand.

When the chained RPCs are terminated by a synchronous-response-type RPC, the

server UAP process can accept the next service request when processing is terminated, provided that the requirements for optimizing transaction are satisfied.

The figures below show the relationship between chained RPCs and the synchronization point.

Figure 2-35: Relationship between chained RPCs and synchronization point (transactional chained RPCs)

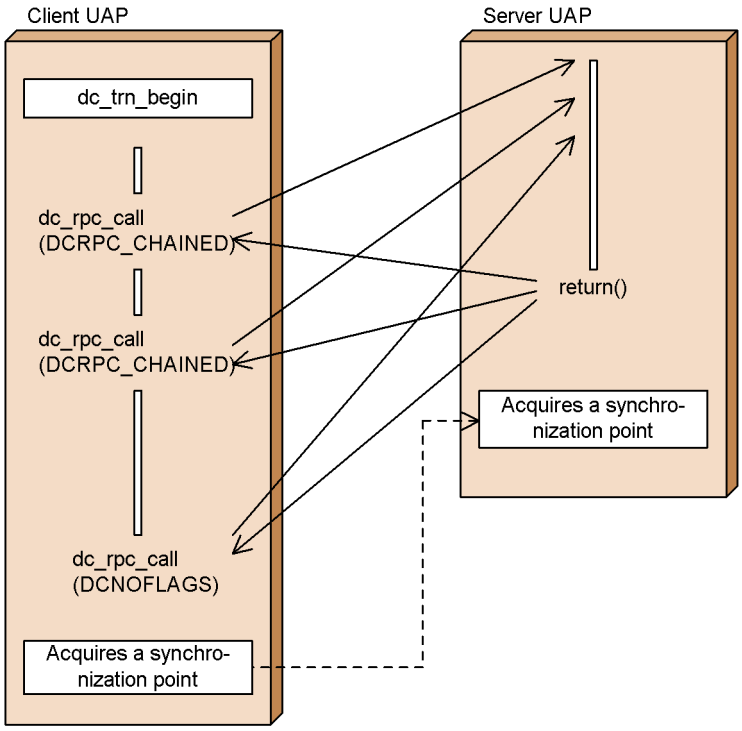
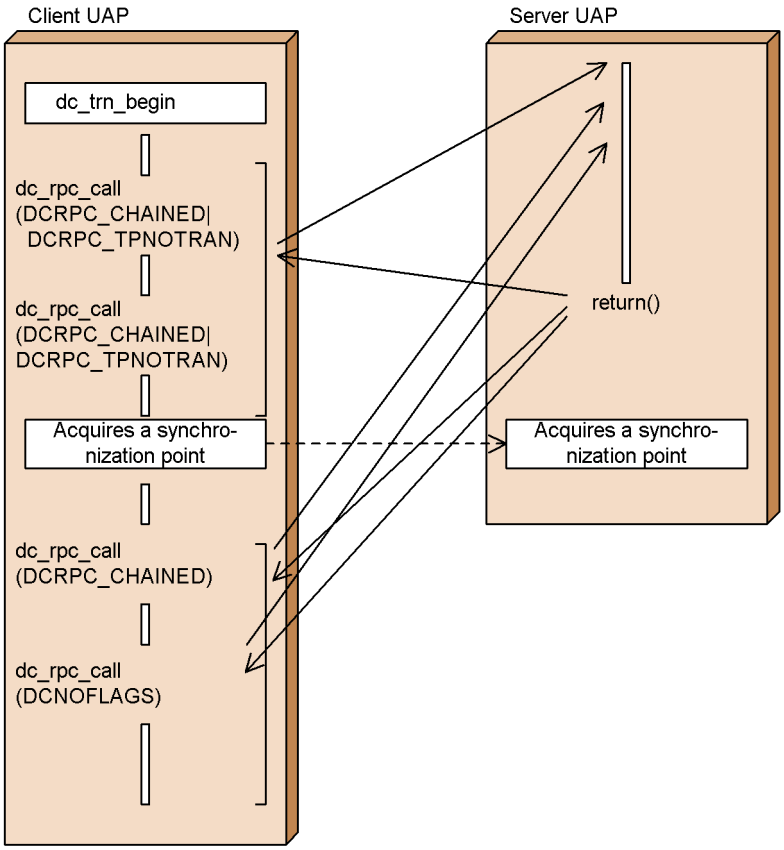


Figure 2-36: Relationship between chained RPCs and synchronization points (if a specification is given so that the server processing will not end with the non-transactional chained RPCs)



(5) RPC error return values and synchronization points

Even when the function `dc_rpc_call()` or the function `dc_rpc_poll_any_replies()` returns with an error, the transaction synchronization point might becomes commitment.

A transaction might have to be rolled back depending on the return value. In this case, use a rollback function (`dc_trn_chained_rollback()` or `dc_trn_unchained_rollback()`) to roll back the transaction.

The following return values cause a transaction to be rolled back:

- Return value of the function `dc_rpc_call()`
- Return value of the function `dc_rpc_poll_any_replies()`

For details on the function `dc_rpc_call()` or `dc_rpc_poll_any_replies()` that must be rolled back, see the applicable *OpenTP1 Programming Reference* manual.

2.3.5 Transaction optimization

OpenTP1 provides the following types of optimization to improve the performance of transaction processing:

- Commit optimization: 1
- Prepare optimization: 2
- Asynchronous prepare optimization: 3
- One-phase optimization: 4
- Read-only optimization: 5
- No-access optimization: 6
- Rollback optimization: 7

There are some conditions for each optimization. Creating UAPs which satisfy the conditions can improve the performance of transaction processing.

The priority of each optimization is as follows:

5, 6, 7 > 2 > 3 > 4 (1 is performed together with other optimization.)

The purpose of transaction optimization is to improve the performance of the synchronization point processing between transaction branches on the client side and the server side. Therefore, multiple types of optimization can be used in one global transaction.

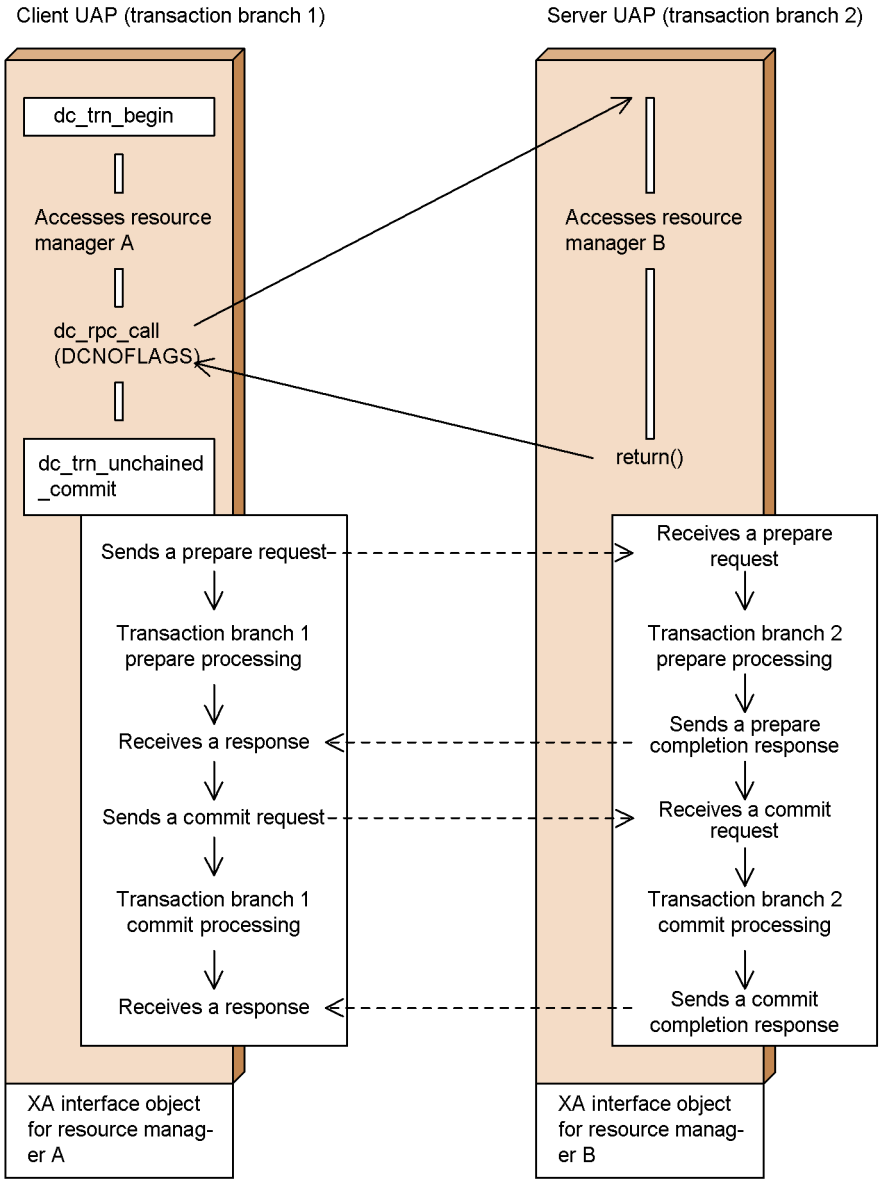
Since chained RPCs reduce the number of transaction branches in a global transaction, they enable transactions to be executed more efficiently.

(1) Ordinary transaction processing (two-phase commit)

OpenTP1 performs control transaction via X/Open XA interface. On XA interface, the synchronization point of transactions is acquired by prepare processing and commit processing separately. This synchronization point processing is called *two-phase commit*. Therefore, the client UAP communicates with the server UAP four times in total: sending two requests for synchronization point processing and receiving two responses. During two-phase commit processing, the process of transaction processing cannot receive other service requests until the synchronization point is acquired.

The figure below shows the outline of ordinary transaction processing (two-phase commit).

Figure 2-37: Outline of ordinary transaction processing (two-phase commit)



(2) Commit optimization

When the conditions for commit optimization are satisfied, the synchronization point processing for phase 2 (commit/rollback processing) to be performed in the transaction branch on the server is performed in the transaction branch on the client. This

eliminates two of the inter-process communications and improves the performance of transaction processing.

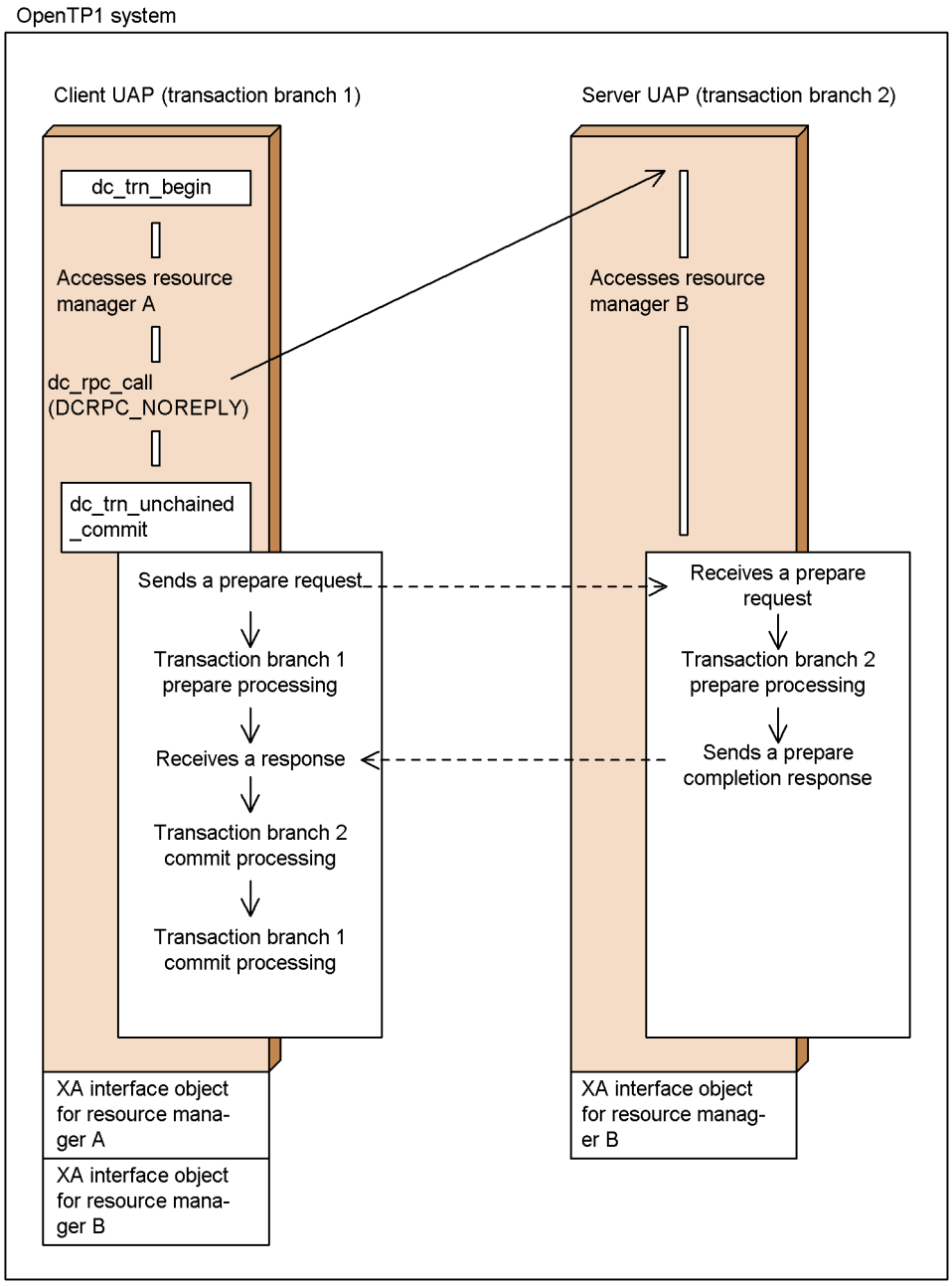
Commit optimization is performed when all the following conditions are satisfied:

1. Both transaction branches on the client and the server are within the same OpenTP1 system.
2. The XA interface object file for the resource manager accessed in the transaction branch on the server has been linked to the transaction branch on the client.

During commit optimization, the transaction branch on the server can receive other service requests when the synchronization point processing for phase 1 terminates, without waiting for completion of processing for phase 2.

The figure below shows the outline of commit optimization.

Figure 2-38: Outline of commit optimization



(3) Prepare optimization

When the conditions for prepare optimization are satisfied, the synchronization point processing for phase 1 (prepare processing) to be performed in the transaction branch on the server is performed in the transaction branch on the client. This eliminates two of the inter-process communications and improves the performance of transaction processing.

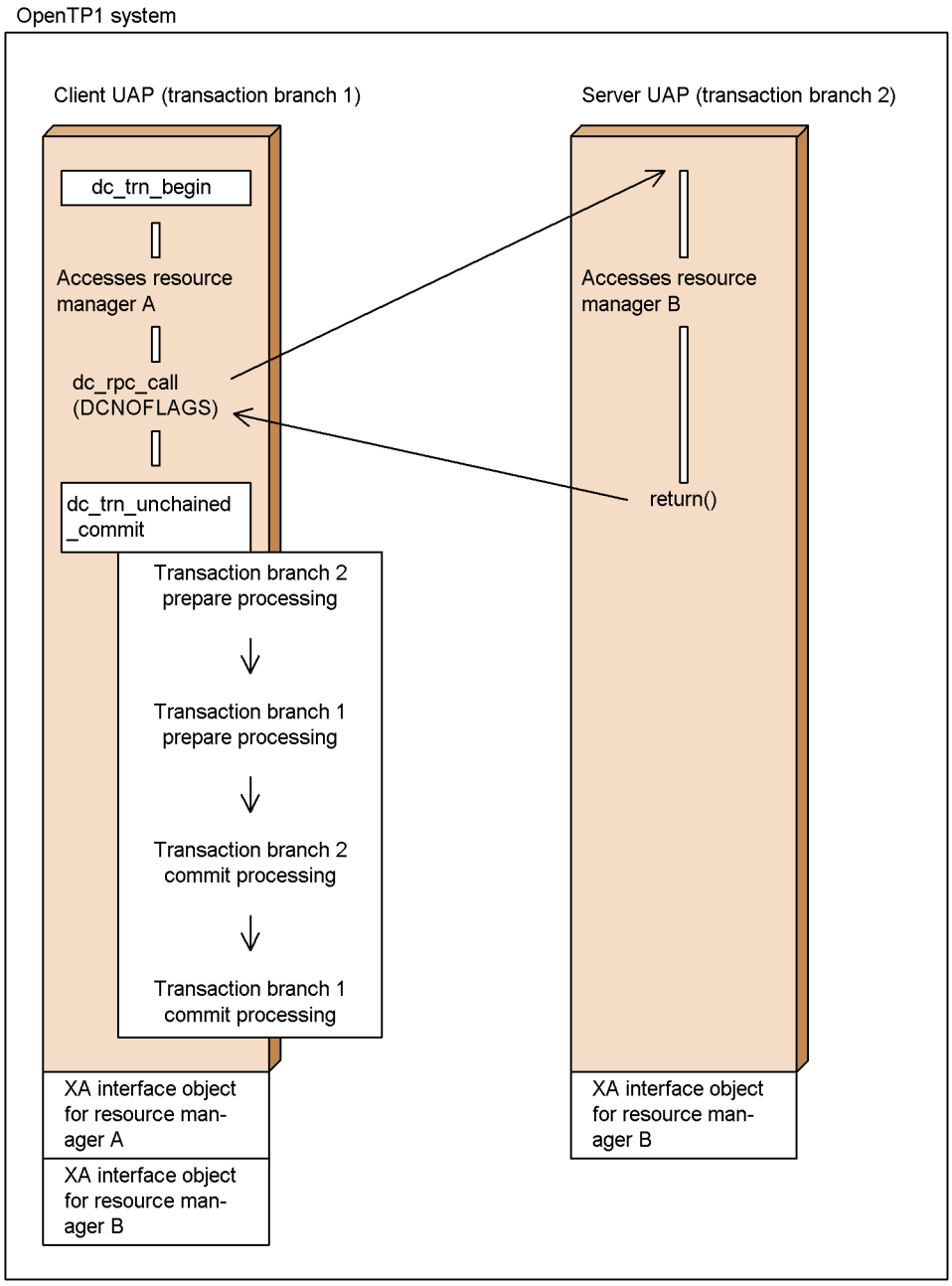
Prepare optimization is performed when all the following conditions are satisfied:

1. Both transaction branches on the client and the server are within the same OpenTP1 system.
2. The XA interface object file for the resource manager accessed in the transaction branch on the server has been linked to the transaction branch on the client.
3. The transaction branch on the client uses synchronous-response RPCs. (`DCNOFLAGS` is specified for `flags` of the function `dc_rpc_call()`.)

Since commit optimization is also performed during prepare optimization, it results in eliminating four of inter-process communications. The transaction branch on the server can receive other service requests without waiting for completion of synchronization point processing.

The figure below shows the outline of prepare optimization.

Figure 2-39: Outline of prepare optimization



(4) Asynchronous prepare optimization

When the conditions for asynchronous prepare optimization are satisfied, the transaction branch on the server performs prepare processing at the time the service processing terminates before control returns to the transaction branch on the client. This eliminates two of the inter-process communications and improves the performance of transaction processing.

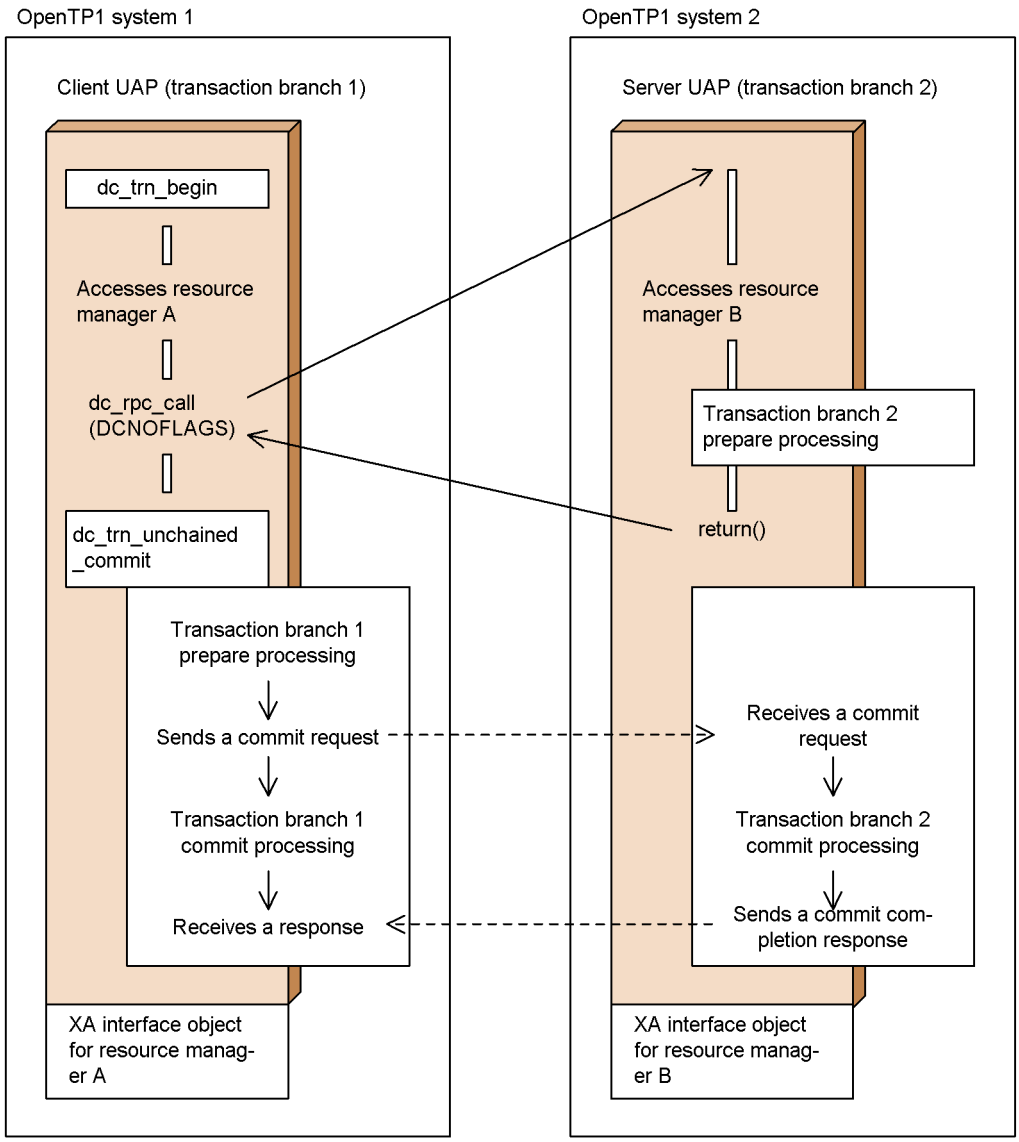
Asynchronous prepare optimization is performed when all the following conditions are satisfied:

1. The UAP on the client specifies asyncprepare in the `trn_optimum_item` operand of the user service definition.
2. Prepare optimization is disallowed. (If allowed, prepare optimization has precedence over asynchronous prepare optimization.)
3. The transaction branch on the client uses synchronous-response RPCs. (`DCNOFLAGS` is specified for `flags` of the function `dc_rpc_call()`.)

Asynchronous prepare optimization has a much longer RPC response time than ordinary transaction processing. If OpenTP1 containing the transaction branch on the client terminates abnormally during transaction processing, OpenTP1 containing the transaction branch on the server may also terminate abnormally due to journal acquisition.

The figure below shows the outline of asynchronous prepare optimization.

Figure 2-40: Outline of asynchronous prepare optimization



(5) One-phase optimization

When the conditions for one-phase optimization are satisfied, only the transaction branch on the server performs the synchronization point processing while the transaction branch on the client does not access the resource manager. This eliminates two of the inter-process communications and improves the performance of transaction

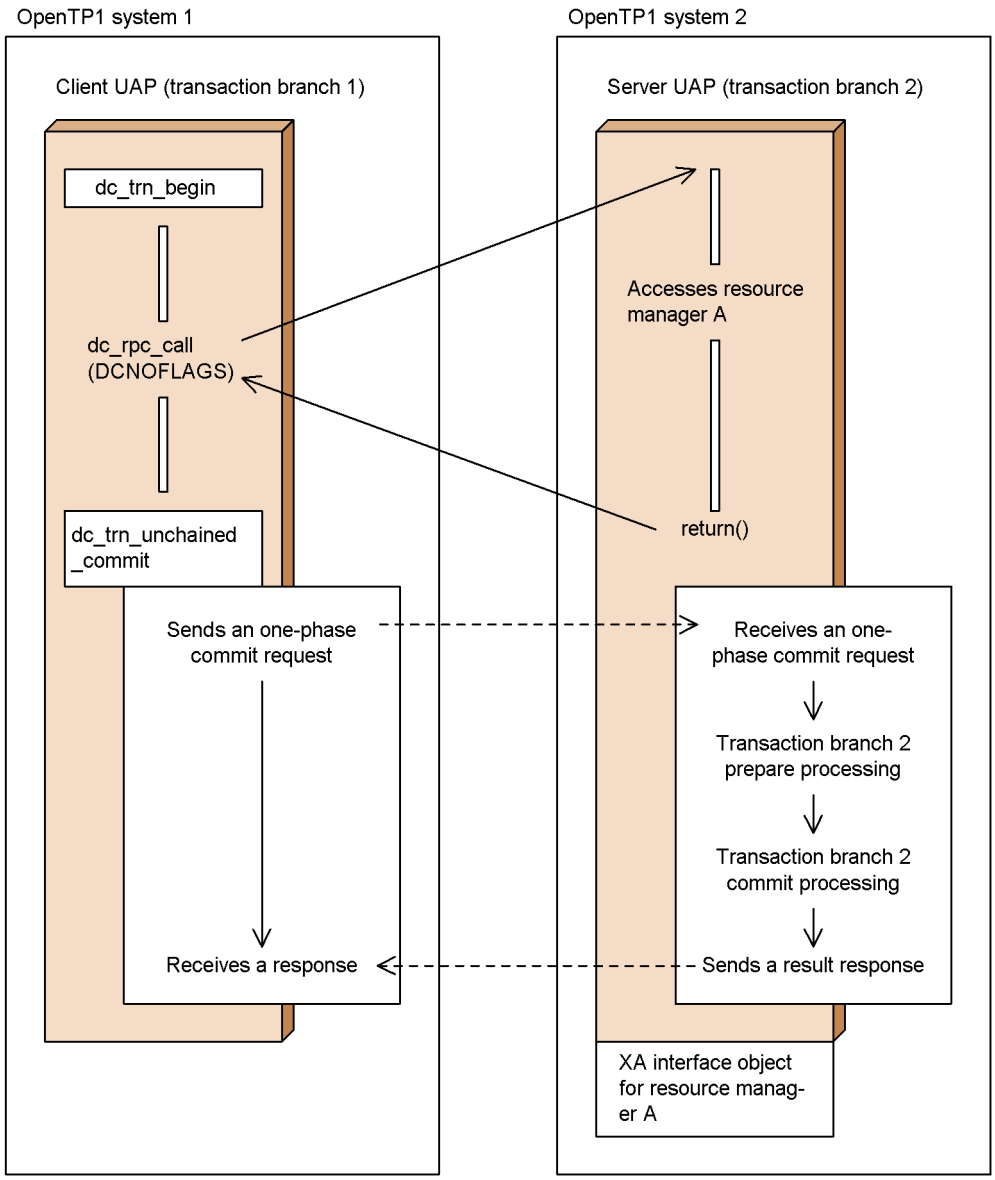
processing.

One-phase optimization is performed when all the following conditions are satisfied:

1. Only the resource manager supporting dynamic registration has been linked to the transaction branch on the client.
2. The transaction branch on the client neither accesses the resource manager nor outputs the user journal.
3. The transaction branch on the client has only one child transaction branch.

The figure below shows the outline of one-phase optimization.

Figure 2-41: Outline of one-phase optimization



(6) Read-only optimization

When the conditions for read-only optimization are satisfied, the synchronization point processing for phase 2 is not performed if the transaction branch on the server does not perform update processing. This eliminates two of the inter-process communications

and improves the performance of transaction processing.

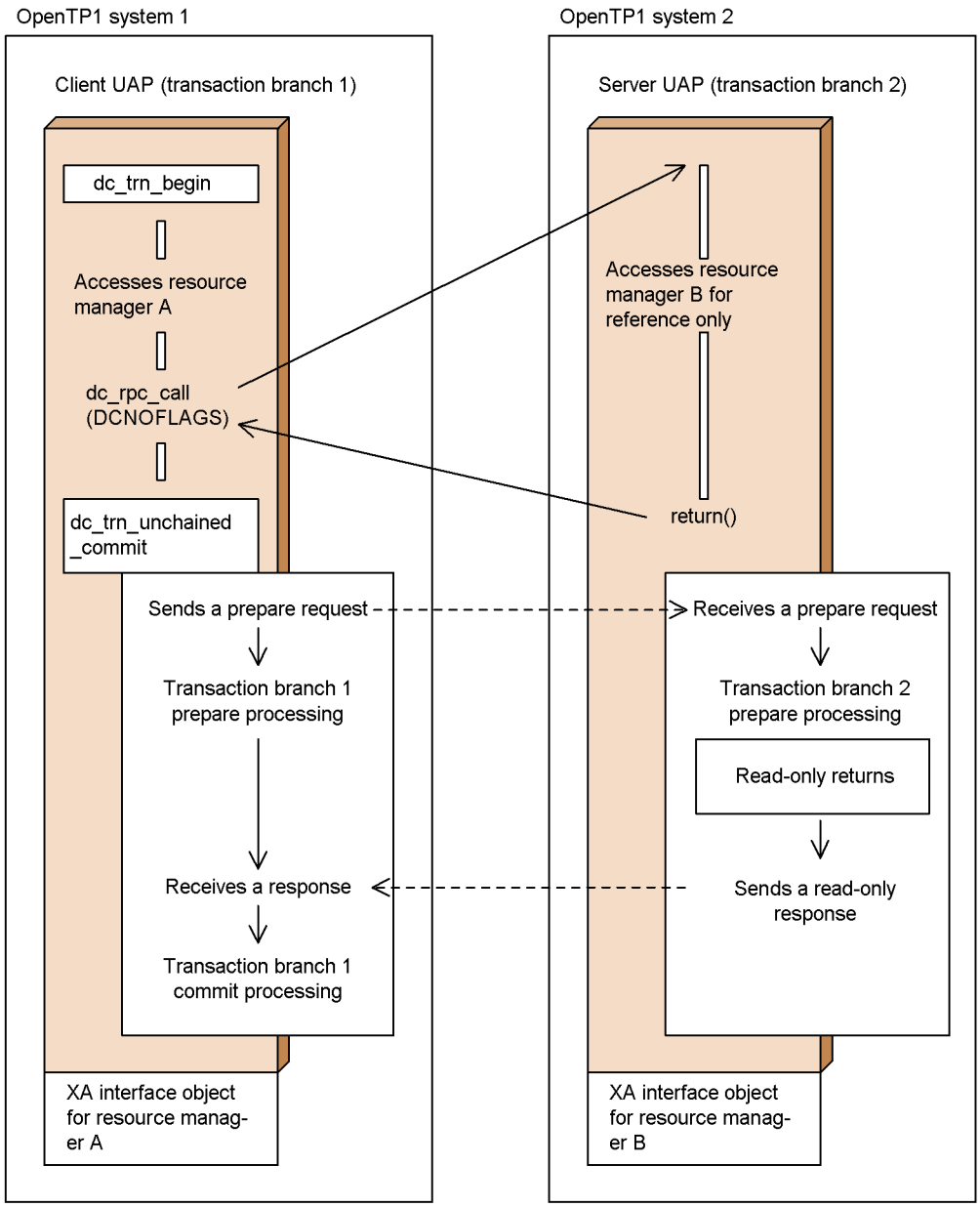
Read-only optimization is performed when all the following conditions are satisfied:

1. The transaction branch on the server neither updates resources (excluding reference) nor outputs the user journal.
2. The transaction branch on the client has only one child transaction branch.

During read-only optimization, the transaction branch on the server can receive the next service request when the synchronization point processing for phase 1 terminates, without waiting for completion of processing for phase 2.

The figure below shows the outline of read-only optimization.

Figure 2-42: Outline of read-only optimization



(7) No-access optimization

When the conditions for no-access optimization are satisfied, the synchronization

point processing is not performed if the transaction branch on the server does not access the resource manager. This eliminates four of the inter-process communications and improves the performance of transaction processing.

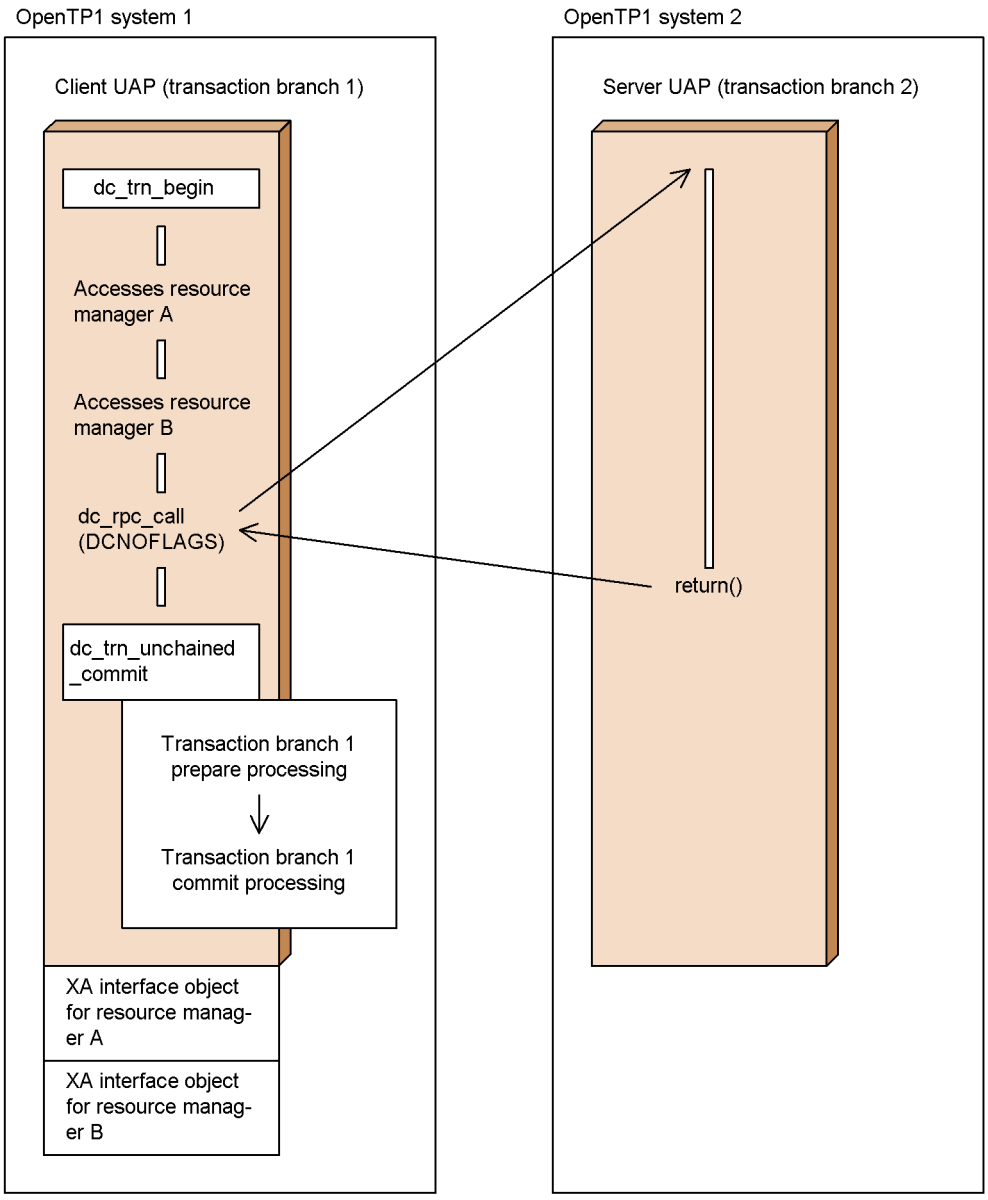
No-access optimization is performed when all the following conditions are satisfied:

1. The transaction branch on the client uses synchronous-response RPCs. (`DCNOFLAGS` is specified for `flags` of the function `dc_rpc_call()`.)
2. Only the resource manager supporting dynamic registration has been linked to the transaction branch on the server.
3. The transaction branch on the server neither accesses the resource manager nor outputs the user journal.
4. The transaction branch on the server has no child transaction branch. Alternatively, it has child transaction branch(es) for which read-only optimization is available.

During no-access optimization, the transaction branch on the server can receive other service requests without waiting for completion of the synchronization point processing.

The figure below shows the outline of no-access optimization.

Figure 2-43: Outline of no-access optimization



(8) Rollback optimization

When the conditions for rollback optimization are satisfied, the transaction branch on the server rolls back out of synchronization with other transaction branches if it uses a rollback function. Other transaction branches do not execute the synchronization point

processing for phase 1. This eliminates two of the inter-process communications and improves the performance of transaction processing.

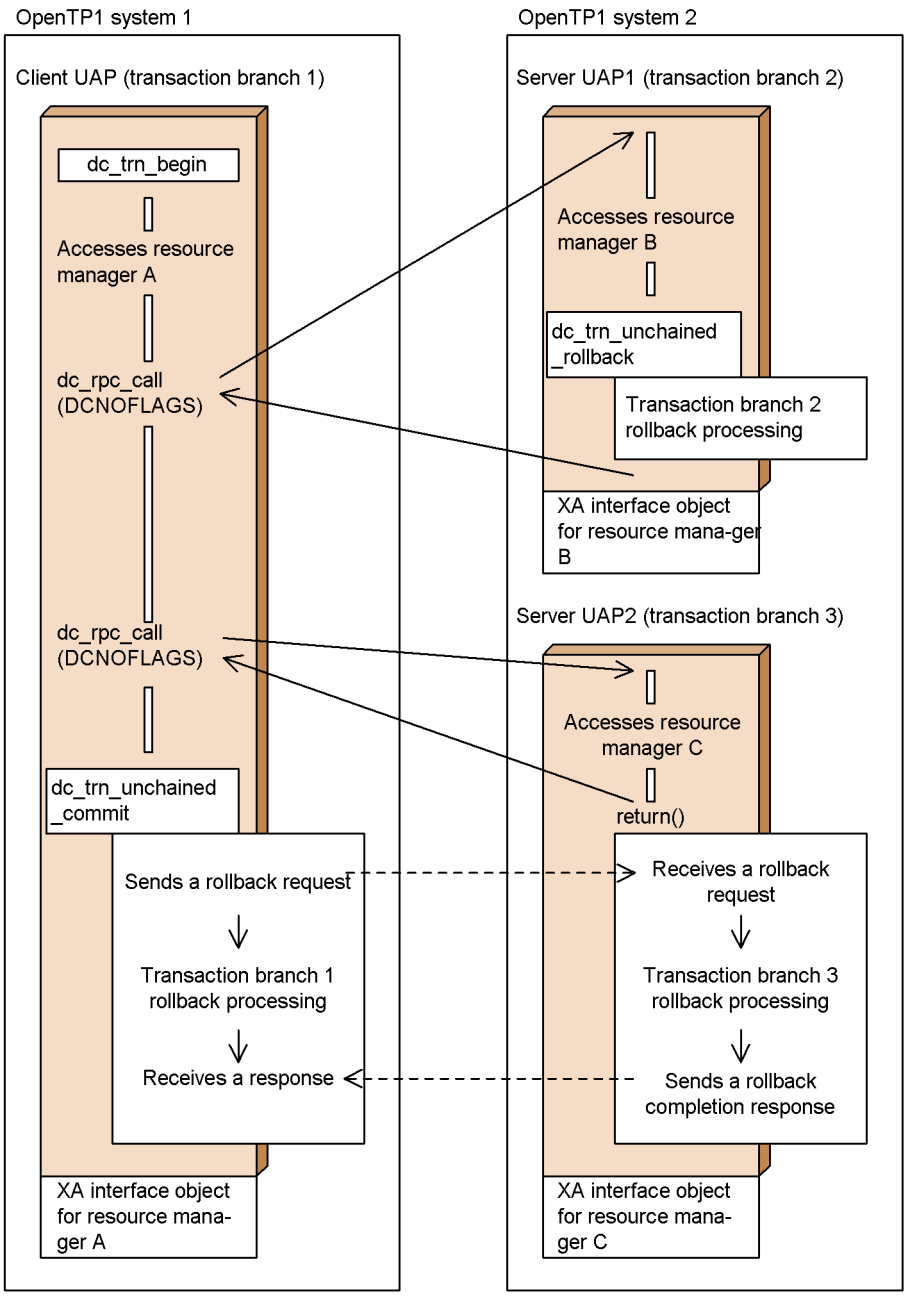
Rollback optimization is performed when all the following conditions are satisfied:

1. The transaction branch on the client uses synchronous-response RPCs. (`DCNOFLAGS` is specified for `flags` of the function `dc_rpc_call()`.)
2. The transaction branch on the server uses a rollback function.

During rollback optimization, the transaction branch on the server can receive other service requests without waiting for completion of the synchronization point processing.

The figure below shows the outline of rollback optimization.

Figure 2-44: Outline of rollback optimization

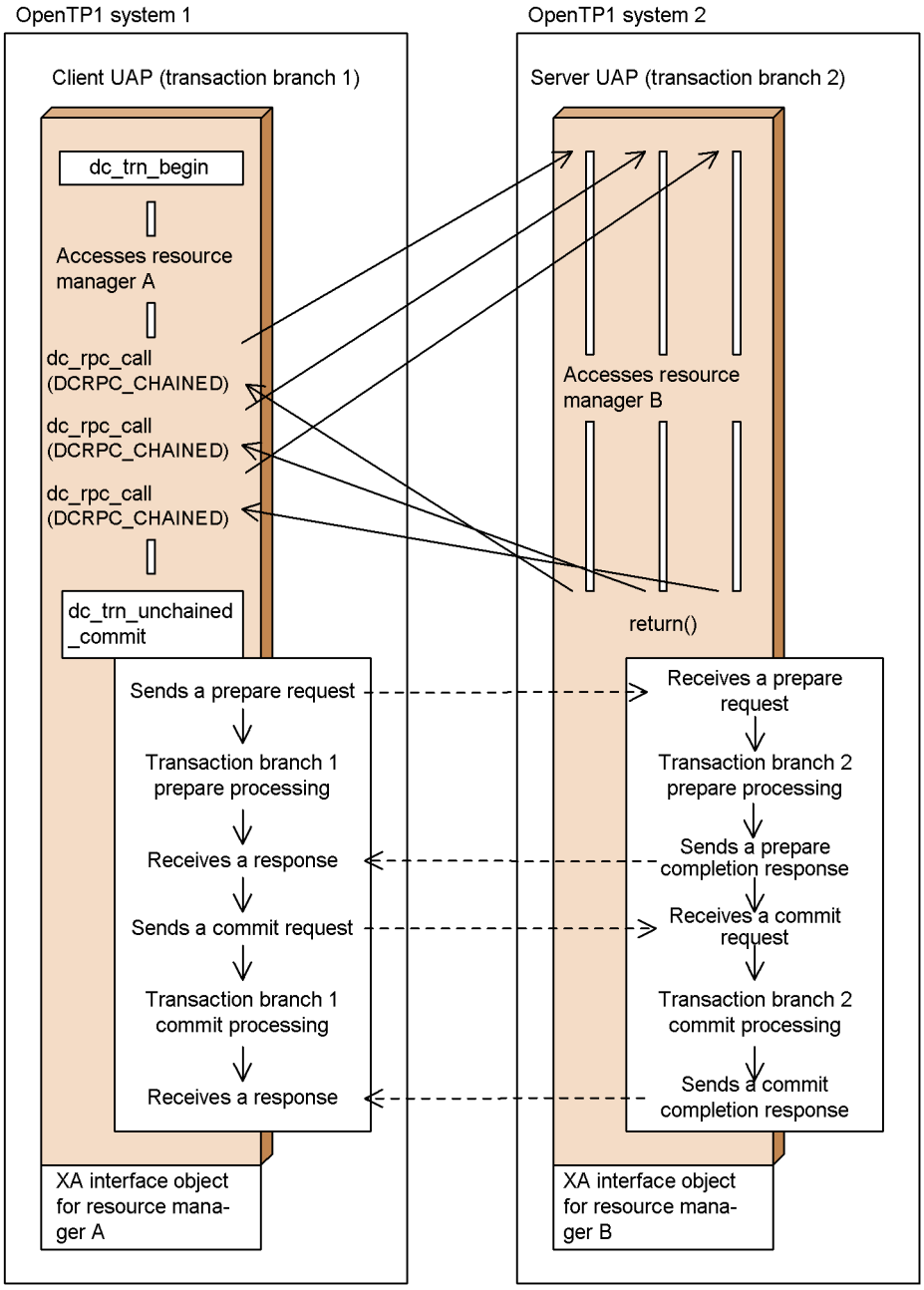


(9) Optimization using chained RPCs

In general, when a service is requested from a transaction branch to a UAP for which the transaction attribute is specified, the process of the transaction branch on the server is handled as another transaction branch. However, when a chained RPC is used for the same service group (various services can exist), that is, when `DCRPC_CHAINED` is specified for `flags` of the function `dc_rpc_call()`, the process is handled as the same transaction branch until the chained RPC is completed. When the conditions for optimization using chained RPCs are satisfied, the performance of transaction processing is improved since the number of transaction branches in a global transaction is reduced.

The figure below shows the outline of optimization using chained RPCs.

Figure 2-45: Outline of optimization using chained RPCs



2.3.6 Posting information about the current transaction

If the function `dc_trn_info()` [`CBLDCTRN('INFO ')`] is used from a UAP, the return value indicates whether the UAP is operating as a transaction.

2.3.7 Disposal in case of heuristic situation

If data cannot be exchanged between transaction branches because a communication error occurred between nodes, the synchronization point must be acquired on each node by executing a command. If the synchronization point is acquired on each node, one transaction branch in the global transaction may be committed and another may be rolled back. Acquiring the synchronization point on each node is called *heuristic decision*. During heuristic decision, a function returns with an error if the synchronization point of the global transaction is acquired from a UAP. One of the following values will be returned from a function due to heuristic determination:

- `DCTRNER_HEURISTIC` (`00903`): The results of heuristic determination did not match the results of the synchronization point of the global transaction.
- `DCTRNER_HAZARD` (`00904`): The results of the synchronization point of the transaction branch that was completed by the heuristic method are unknown.

The results of the synchronization point of the UAP, resource manager, or global transaction that caused the return value to arise can be checked by reading the contents of the message log file.

2.3.8 Notes on transaction processing

(1) Relationship between transaction processing and the user service definition

Note the following points when requesting a service for which the transaction attribute is defined (`atomic_update=Y` specified) from a service that is being executed as a transaction:

1. Specify a sufficient number of processes for the maximum number of processes (`parallel_count`) in the user service definition of the server UAP. Even after server UAP processing terminates, no service is provided to another client UAP until synchronization point processing of the global transaction is completed (unless the optimizing transaction is enabled). If a transaction continues for a long time in such a situation, processes are occupied which are equivalent to different client UAPs that requested services during the transaction. As a result, transaction performance might decline.
2. Depending on the value given to `balance_count` (the number of remaining service requests) in the user service definition, even a user server which uses the multiserver facility may encounter a RPC timeout without any increase in the number of nonresident processes. Specify in the operand `balance_count` the most suitable value considering the load on the server UAP.

   In the following cases, be sure to specify `0` in the `balance_count` operand:

- A recursive call is used with a user server comprising only nonresident processes (e.g., when `parallel_count = 0` or `2`).
- A recursive call is used with a server comprising one resident process and nonresident processes (e.g., when `parallel_count = 1` or `2`).

(2) Time monitoring of transaction processing

As for time monitoring from the transaction start to synchronization point processing, you can specify whether to include the time until the function `dc_rpc_call()` called in the transaction returns. Define this specification with `trn_expiration_time_suspend` of the user service definition, the user service default definition, and the transaction service definition.

For details on the value to be assigned to `trn_expiration_time_suspend` and transaction time monitoring, see the manual *OpenTP1 System Definition*.

2.4 System operation management

This section explains how to execute OpenTP1 commands by invoking functions from within a UAP, how to report start processing completion, and how to obtain the UAP status by invoking an appropriate function from within the UAP.

2.4.1 Executing operation commands

To support OpenTP1 system operation, you can execute commands, which can be entered in online mode, from a UAP by using the function `dc_adm_call_command()` [`CBLDCADM('COMMAND ')`]. The execution results of commands are returned to the UAP. The results comprise values which are output to the standard output or standard error output.

Give the following specification to the UAP which is to execute commands, in order to define the directory containing the command as the command search path:

- TP1/Server Base:

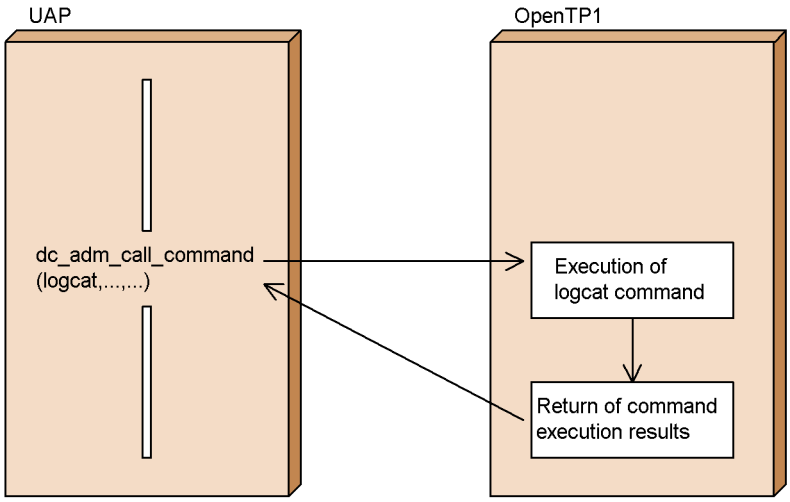
  Specify the environment variable for `putenv PATH` in the user service definition.

- TP1/LiNK:

  Add a search path when setting up a TP1/LiNK environment.

The figure below outlines OpenTP1 command execution using the function `dc_adm_call_command()`.

Figure 2-46: Outline of OpenTP1 command execution using function dc_adm_call_command()



(1) OpenTP1 commands which can be executed using the function dc_adm_call_command()

Table 2-1 lists the OpenTP1 commands and indicates which commands can be executed from UAPs. For details of results of command input which come from OpenTP1, see the manual *OpenTP1 Operation*.

Table 2-1: OpenTP1 commands which can be executed from UAPs

Facility		Command name	Can/cannot be executed from UAPs
System management	Catalog OpenTP1 into OS, delete OpenTP1 from OS	`dcsetup`	N
	Restart process service and reflect definitions	`dcreset`	N
	Reserve or release resources for OpenTP1 internal control	`dcmakeup`	N
	Start OpenTP1	`dcstart`	N
	Terminate OpenTP1	`dcstop`	Y#
	Output system statistical information	`dcstats`	Y
	Start multinode area/subarea	`dcmstart`	Y
	Terminate multinode area/subarea	`dcmstop`	Y
	Execute OpenTP1 commands from the scenario template	`dcjcmdex`	N
	Specify an operand of the system definition	`dcjchconf`	N
	Update the domain definition file	`dcjnamch`	Y
	Display the status of OpenTP1 node	`dcndls`	Y
	Display shared memory utilization status	`dcshmls`	Y
	Display execution status of temporary close processing	`rpcstat`	Y
	Redirect standard output and standard error output	`prctee`	N
	Stop and restart the `prctee` process	`prctctrl`	N
	Acquire maintenance documents	`dcrasget`	Y
	Edit and output system statistical information to the standard output in real time	`dcreport`	Y

Facility		Command name	Can/cannot be executed from UAPs
	Delete troubleshooting information	dccspool	Y
	Check the system definition	dcdefchk	N
	Display product information	dcpplist	Y
Remote API management	Display the RAP-processing listener or RAP-processing server status.	rapls	N
	Set up the execution environment of a remote API facility	rapsetup	N
	Generate automatically definitions used for a remote API facility	rapdfgen	N
Server management	Start server	dcsvstart	Y
	Terminate server	dcsvstop	Y
	Display status of server	prcls	Y
	Display search path names for user server and for command activated from user server	prcpathls	Y
	Change search path names for user server and for command activated from user server	prcpath	Y
	Abort OpenTP1 process	prckill	Y
Schedule management	Display scheduling status	scdls	Y
	Shut down the scheduling	scdhold	Y
	Restart scheduling	scdrles	Y
	Change the number of processes	scdchprc	Y
	Stop and restart a process	scdrsprc	Y
Transaction management	Display status of transactions	trnls	Y
	Commit transactions	trncmt	Y
	Roll back transactions	trnrbk	Y
	Terminate transactions forcibly	trnfgt	Y
	Start and terminate collecting of transaction statistical information	trnstics	Y

Facility		Command name	Can/cannot be executed from UAPs
	Delete undecided transaction information file	trndlinf	Y
	Display undecided transaction information for OSI TP communication	tptrnls	Y
XA resource management	Display an XAR event trace information	xarevtr	N
	Display status of an XAR file	xarfills	Y
	Change status of an XAR transaction	xarforce	Y
	Shut down an XA resource service	xarhold	Y
	Create an XAR file	xarinit	N
	Display an XAR transaction information	xarls	Y
	Release an XA resource service from shutdown	xarrles	Y
	Delete an XAR file	xarrm	N
Exclusion management	Display lock information	lckls	Y
	Display lock table pool information	lckpool	Y
	Delete deadlock information file or timeout information file	lckrminf	Y
Name management	Check OpenTP1 startup and delete a cache	namalivechk	Y
	Catalog and delete domain alternate schedule service	namdomainsetup	Y
	Change domain configuration (using the system common definition)	namndchg	Y
	Change domain configuration (using the domain definition file)	namchgfl	Y
	Perform a forced invalidation of the startup notice information	namunavl	N
	Display server information about OpenTP1	namsvinf	Y
	Manipulate the RPC suppression list	namblad	Y
Message log management	Display message log file	logcat	Y
	Switch message log realtime output function	logcon	Y

Facility		Command name	Can/cannot be executed from UAPs
Audit logs	Set up the environment for audit logging	dcauditsetup	N
OpenTP1 file management	Initialize a OpenTP1 file system	filmkfs	N
	Display status of an OpenTP1 file system	filstatfs	Y
	Display contents of an OpenTP1 file system	fills	Y
	Back up an OpenTP1 file system	filbkup	N
	Restore an OpenTP1 file system	filrstr	N
	Change an OpenTP1 file group	filchgrp	Y
	Change an OpenTP1 file access authorization mode	filchmod	Y
	Change an OpenTP1 file owner	filchown	Y
Status file management	Create and initialize a status file	stsinit	N
	Display status of status files	stsls	Y
	Display contents of a status file	stsfills	Y
	Open a status file	stsopen	Y
	Close a status file	stsclose	Y
	Delete a status file	stsrm	Y
	Swap status files	stsswap	Y
Journal file management	Initialize a journal file	jnlinit	N
	Display journal file information	jnlls	Y
	Display information about previously read journal files during a rerun	jnlrinf	N
	Open a journal file	jnlopnfg	Y
	Close a journal file	jnlclsfg	Y
	Allocate journal physical file	jnladdpf	Y
	Delete journal physical file	jnldelpf	Y
	Allocate a journal file dynamically	jnladdpf	Y
	Swap journal files	jnlswpfg	Y

Facility		Command name	Can/cannot be executed from UAPs
	Delete journal files	jnlrm	N
	Change status of journal files	jnlchgfg	N
	Unload journal files	jnlunlfg	N
	Control the automatic unload facility	jnlatunl	N
	Recover journal files	jnlmkrf	N
	Integrate file recovery journals	jnlcolc	N
	Copy unload journal files	jnlcopy	N
	Display archive status	jnlarls	Y
	Edit and output unload journal files or global archive unload journal files	jnledit	N
	Output records from unload journal files or global archive unload journal files	jnlrput	N
	Sort and merge unload journal files or global archive unload journal files chronologically	jnlsort	N
	Output uptime statistical information	jnlstts	N
	Output MCF uptime statistical information	jnlmcst	N
	Forcibly release connection to resource group	jnlardis	N
DAM file management	Initialize a physical file	damload	N
	Display status of logical files	damls	Y
	Add a logical file	damadd	Y
	Remove a logical file	damrm	Y
	Shut down a logical file logically	damhold	Y
	Release logical file from the shutdown	damrles	Y
	Delete a physical file	damdel	N
	Back up a physical file	dambkup	N
	Restore a physical file	damrstr	N
	Recover a logical file	damfrc	N

Facility		Command name	Can/cannot be executed from UAPs
	Set a threshold for the number of cache blocks	damchdef	Y
	Obtain the number of cache blocks	damchinf	Y
TAM file management	Initialize a TAM file	tamcre	N
	Display status of TAM tables	tamls	Y
	Add a TAM table	tamadd	Y
	Remove a TAM table	tamrm	Y
	Shut down a TAM table logically	tamhold	Y
	Release a TAM table from shutdown	tamrles	Y
	Load a TAM table	tamload	Y
	Unload a TAM table	tamunload	Y
	Delete a TAM file	tamdel	N
	Back up a TAM file	tambkup	N
	Restore a TAM file	tamrstr	N
	Recover a TAM file	tamfrc	N
	Convert a TAM locked resource name	tamlckls	Y
	Display synonym information about hash type TAM files and TAM tables	tamhsls	N
Message queue file management	Display status of queue groups	quels	Y
	Allocate physical file for message queue	queinit	N
	Delete physical file for message queue	querm	N
Resource manager control	Display resource manager information	trnlsrm	N
	Catalog and delete the resource manager	trnlnkrm	N
	Create a transaction control object file	trnmkobj	N
Trace management	Output UAP trace information	uatdump	N
	Merge RPC traces	rpcmrg	N
	Output RPC trace information	rpcdump	N

Facility		Command name	Can/cannot be executed from UAPs
	Output shared memory dump	usmdump	Y
Management of performance verification traces	Edit and output trace information file	prfed	N
	Get trace information file	prfget	N
Real-time statistical information service management	Edit and output RTS log files	rtsedit	N
	Output real-time statistical information to the standard output	rtsls	N
	Set up an execution environment for the real-time statistical information service	rtssetup	N
	Change the settings for real-time statistical information	rtsstats	N
Connection management	Display status of connection	mcftlscn	Y
	Establish connections	mcftactcn	Y
	Release connections	mcftdctcn	Y
	Switch connections	mcftchcn	Y
	Display network status	mcftlsln	Y
	Start acceptance of server-type connection establishment requests	mcftonln	Y
	Terminate acceptance of server-type connection establishment requests	mcftofln	Y
	Display status of multiplex message processing	mcftlstrd	Y
Application management	Display status of applications	mcfalsap	Y
	Shut down applications	mcfadctap	Y
	Release shutdown of applications	mcfaactap	Y
	Initialize abnormal terminations counts applications	mcfaclcap	Y
	Display status of application start request	mcfalstap	Y
	Delete timer activation requests for applications	mcfadltap	Y
Application operation support	Start application programs	mcfuevt	Y

Facility		Command name	Can/cannot be executed from UAPs
Logical terminal management	Display status of logical terminals	mcftlsle	Y
	Shut down logical terminals	mcftdctle	Y
	Release shutdown of logical terminals	mcftactle	Y
	Skip first message in a logical terminal message queue	mcftspqle	Y
	Hold process of a logical terminal output queue	mcfthldoq	Y
	Release held process of a logical terminal output queue	mcftrlsoq	Y
	Delete output queues for logical terminals	mcftdlqle	Y
	Start message journal collection for logical terminals	mcftactmj	Y
	Terminate message journal collection for logical terminals	mcftdctmj	Y
	Terminate forcibly continuous-inquiry-response processing for logical terminals	mcftendct	Y
	Start the alternate terminal	mcftstalt	Y
	Terminate the alternate terminal	mcftedalt	Y
Service group management	Display status of service groups	mcftlssg	Y
	Shut down service groups	mcftdctsg	Y
	Release service groups from shutdown	mcftactsg	Y
	Hold process of input queue for service group	mcfthldiq	Y
	Release held process of input queues for service group	mcftrlsiq	Y
	Delete the input queue for a service group	mcftdlqsg	Y
Service management	Display status of services	mcftlssv	Y
	Shut down services	mcftdctsv	Y
	Release services from shutdown	mcftactsv	Y

Facility		Command name	Can/cannot be executed from UAPs
Session management	Start a session	mcftactss	Y
	Terminate a session	mcftdctss	Y
Buffer management	Display utilization status of buffer groups	mcftlsbuf	Y
Map management	Change path name of a map file	dcmapchg	N
	Display the loaded resources in the map file	dcmapls	N
Queue management	Output contents of input/output queues	mcftdmpqu	Y
MCF trace acquisition management	Swap MCF trace files forcibly	mcftswptr	Y
	Start MCF trace acquisition	mcftstrtr	Y
	Terminate MCF trace acquisition	mcftstptr	Y
Management of MCF statistics	Edit MCF statistics	mcfreport	N
	Output MCF statistics	mcfstats	Y
MCF communication service management	Partially stop the MCF communication service	mcftstop	N
	Partially start the MCF communication service	mcftstart	N
	Reference the status of the MCF communication service	mcftlscom	N
User timer monitoring	Display status of user timer monitoring	mcftlsutm	Y

Legend:

Y: Can be executed from UAPs.

N: Cannot be executed from UAPs.

#

When the dcstop command is executed from UAPs, it should be in the background.

2.4.2 Reporting completion of user server start processing

The function dc_adm_complete() [CBLDCADM('COMPLETE')] for reporting the completion of user server start processing must be called for SUPs. After using the function dc_rpc_open() (which starts UAPs) to OpenTP1, call the function dc_adm_complete() to report the completion of start processing to OpenTP1.

SPPs and MHPs assume that start processing is completed when the function

`dc_rpc_mainloop()` or the function `dc_mcf_mainloop()` is executed normally. Thus, there is no need to use the function `dc_adm_complete()` for SPPs and MHPs.

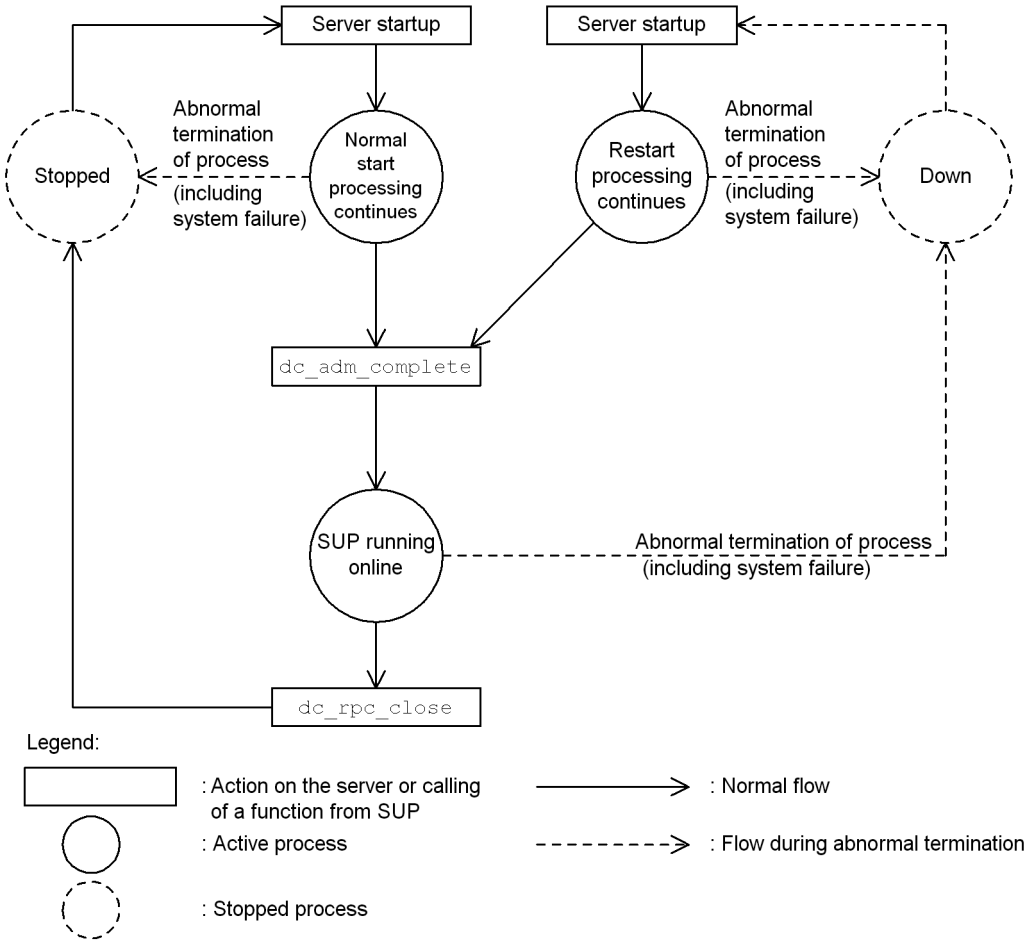
The function `dc_adm_complete()` cannot be used from UAP that handles offline work.

2.4.3 Detecting the user server status

The status of a user server (e.g., whether the user server is active) can be obtained through a UAP. OpenTP1 returns the user server status when the function `dc_adm_status()` [`CBLDCADM('STATUS  ')`] is called from the UAP.

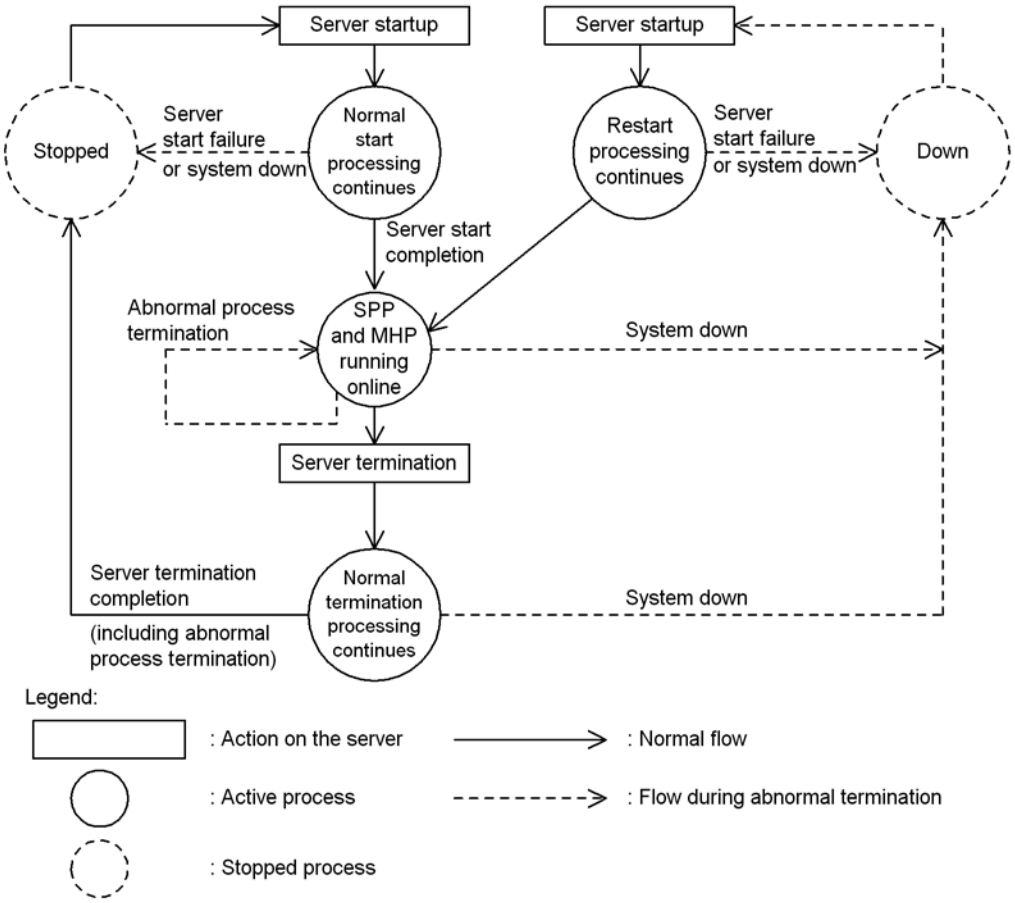
Figures 2-47 to 2-49 show the transition of user server status. The status of the servers shown in the figures is returned from OpenTP1.

Figure 2-47: Transition of user server status (SUP)



#1: When OpenTP1 is started with forced normal start (`dcstart -n` command), a normal start occurs regardless of the previous user server status.
#2: The user server is started with server start (`dcsvstart` command), OpenTP1 system start (`dcstart` command), or the OpenTP1 system's automatic start facility.
#3: When the user server is forcibly stopped, the same transition occurs as that which occurs during abnormal process termination, including abnormal termination of the OpenTP1 system.

Figure 2-48: Transition of user server status (SPP, MHP)



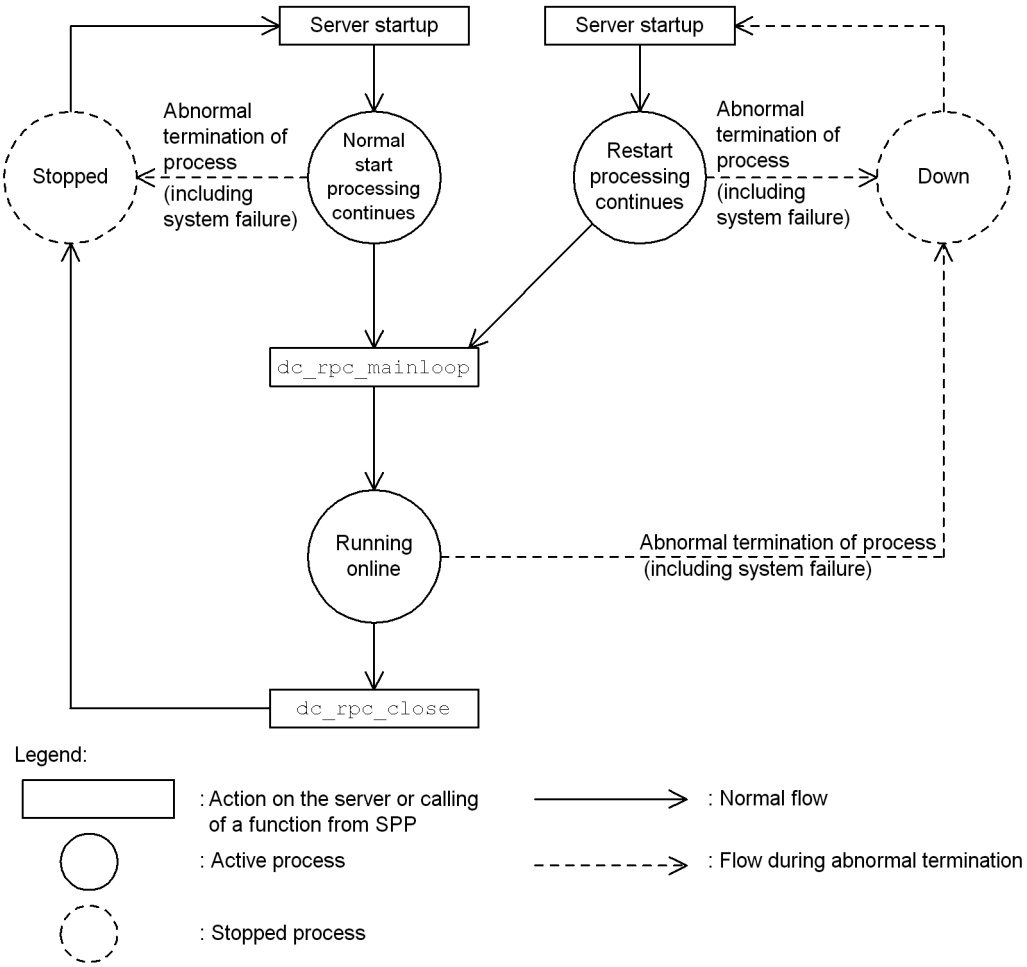
#1: When OpenTP1 is started with forced normal start (`dcstart -n` command), a normal start occurs regardless of the previous user server status.

#2: The user server is started with server start (`dcsvstart` command), OpenTP1 system start (`dcstart` command), or the OpenTP1 system's automatic start facility.
During this process, the number of processes designated as resident processes are started, and when the function `dc_rpc_mainloop` or `dc_mcf_mainloop` has executed normally for all processes, server start processing is completed.
If there is any process that terminates before calling of the function `dc_rpc_mainloop` or `dc_mcf_mainloop` succeeds, server start processing fails.

#3: The user server is terminated with server termination (`dcsvstop` command) or OpenTP1 system termination (`dcstop` command).

#4: When the user server is forcibly stopped, the same transition occurs as that which occurs during abnormal process termination, including abnormal termination of the OpenTP1 system.

Figure 2-49: Transition of user server status (server that receives requests from socket (SPP))



#1: When OpenTP1 is started with forced normal start (`dcstart -n` command), a normal start occurs regardless of the previous user server status.
#2: The user server is started with server start (`dcsvstart` command), OpenTP1 system start (`dcstart` command), or the OpenTP1 system's automatic start facility.
#3: When the user server is forcibly stopped, the same transition occurs as that which occurs during abnormal process termination, including abnormal termination of the OpenTP1 system.

2.5 Message log output

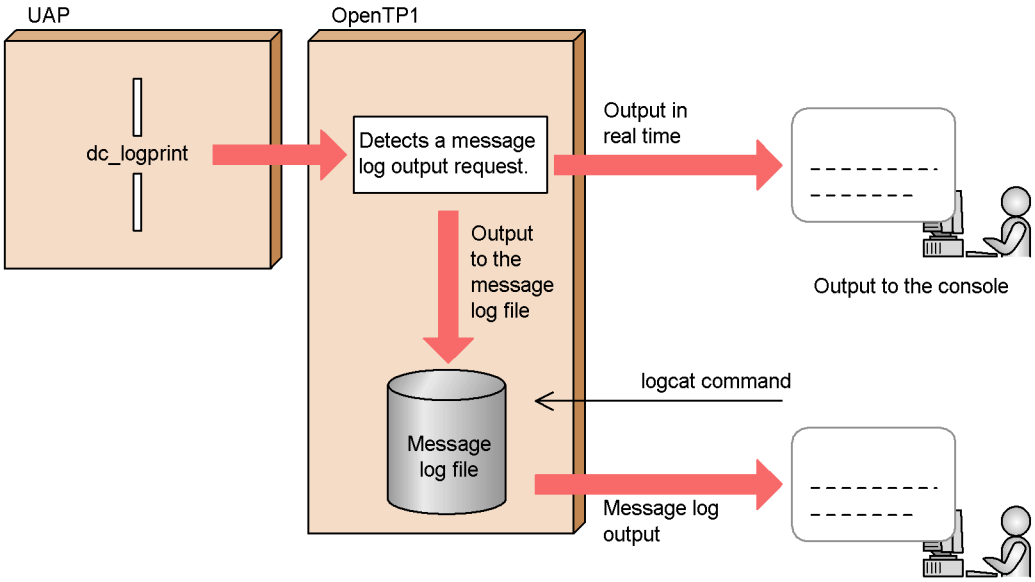
2.5.1 Outputting message log from application programs

User-selected information can be output as a message log from OpenTP1 when the function `dc_logprint()` [`CBLDCLOG('PRINT  ')`] is called from the UAP. The message log is output to the message log file. To display the contents of the message log file, execute the `logcat` command to output the contents to the standard output.

The message log can also be output to the standard output in real time when it is output to the message log file. Whether to output the message log to the standard output in real time can be specified in the log service definition.

The figure below shows message log output from a UAP.

Figure 2-50: Outline of message log output from UAP



(1) Contents of output message logs

Table 2-2 explains the contents of message log information to be output to the message log file. The request source program ID, message ID, and message log text are the items to be specified from the UAP. The information shown in Table 2-2 and OpenTP1 control codes are output to the message log file.

Table 2-2: Contents of message logs output to message log file

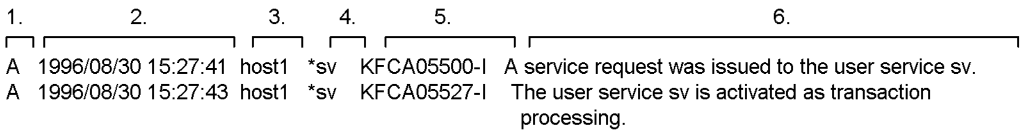
Item number	Item		Output length	Explanation
--	Line header	Message log serial numbers	7 single-byte characters	Serial numbers of all message logs. If a message log is missing due to an error, the message log is identified because the serial numbers do not include the corresponding message log serial number.
		Process ID	5 single-byte characters	ID of the process that specified message log output.
		Message log serial number for each process	7-digit single-byte number	Message log serial number for each process that requested output.
(1)	OpenTP1 ID		2 single-byte alphanumeric characters	OpenTP1 system ID
(2)	Date and time		19-digit integer	Output request time of the message log. The message log is output in the *year*/*month*/*date hour*:*minute*:*second* format.
(3)	Request source node name		8 single-byte alphanumeric characters	Name of the node having the UAP that requested message log output. The first 8 characters of the node name are output.
(4)	Request source program ID		3 single-byte alphanumeric characters	The first character is fixed as *. Two characters specified as the request source program ID in the UAP are set following *.

Item number	Item	Output length	Explanation
(5)	Message ID	11 single-byte alphanumeric characters	ID given to each message log by a UAP when the UAP requested message log output. The message ID format is as follows: `KFCA`$n_1n_2n_3n_4n_5$-*X* `KFCA`: Fixed part $n_1n_2n_3n_4n_5$: Serial number specified in the UAP. Serial numbers `05000` to `06999` are assigned to message logs output by the UAP. *X*: Uppercase letter which indicates the type of the message log `E`: Error message log `I`: Informational message log `W`: Warning message log `R`: Response request message log
(6)	Message log text	Variable length up to 222 characters	Character string in shift JIS code specified by the UAP.

(2) Output format of message logs

The following figure shows the display format of message log data output from a UAP by the function `dc_logprint()` and viewed in the standard output using the `logcat` command. The command options are omitted in this example. For details on the `logcat` command, see the manual *OpenTP1 Operation*. The circled numbers in the figure correspond to the numbers in Table 2-2.

Figure 2-51: Output format of message logs



1. OpenTP1 ID
2. Date and time
3. Request source node name
4. Request source program ID
   "*" at the beginning indicates that the UAP requested the output of the message log.
5. Message ID
   Serial numbers 05000 to 06999 are assigned to message logs which were requested to be output by the UAP
6. Message log text
   Character string specified by the UAP

(3) Notes on passing messages to NETM

Message logs issued by UAPs can be output to the operation support terminal of the integrated network management system (NETM) in the same manner as for OpenTP1 message logs. The contents of a message log output to the NETM include the following information:

- Item in the line header (specified in the log service definition)
- Message ID
- Message text

The display color of a message log to be output to the operation support terminal can be specified in the UAP.

Note the following points when outputting a message log output by a UAP to the NETM:

- The message log to be output to the NETM must be 160 bytes or less. If the message log exceeds 160 bytes, the NETM divides the message when passing it to VOS3. Consequently, another message might be inserted between lines of one message upon output. Also, if the message log exceeds 256 bytes, the OpenTP1 log service truncates the excess bytes.
- Do not include return character \n in the message text to be output to the NETM. If the message text includes \n, the NETM divides the message at \n when passing the message to VOS3. Consequently, another message might be inserted between lines of one message upon output.

NETM: Integrated Network Management System

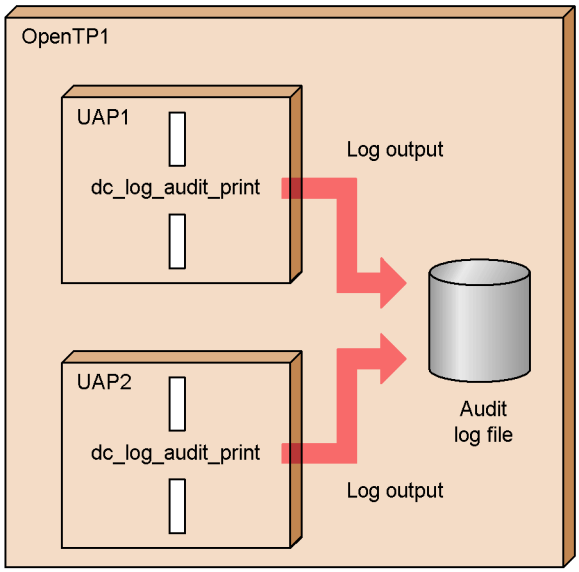
2.6 Audit log output

An audit log is a file containing historical information about the operations performed on OpenTP1 programs by system developers, operators, and users, together with the program behavior triggered by those operations.

In OpenTP1, an entry is output to the audit log when an operation is executed on a UAP, or when internal processing takes place in a UAP. To acquire user-specified audit log data, call the function `dc_log_audit_print()` from the UAP.

The figure below shows the flow of collecting an audit log from UAPs.

Figure 2-52: Outline of audit logging from UAPs



The table below lists the items entered in an audit log file and describes their content. The items that can be specified from the UAP are the message ID, source component, event type, event result, action information, and comment.

Table 2-3: Items output to audit log file

Specified by	Item	Output length (max. bytes)	Description
Items specifiable from a UAP	Message ID	11	The ID of the audit log entry
	Source component	3	The name of the component in which the event occurred. The source component is output in the format **AA*, where *AA* is the value specified by the function `dc_log_audit_print()`.
	Event type	32	The event category
	Event result	10	The result of the event
	Action information	32	The action initiated on the object by the subject who caused the event (Refer/Add/Update/Delete etc.)
	Comment	1024	A comment describing the nature of the event
Items specified automatically by OpenTP1	Header information	12	Header information in the audit log
	Sequence number	7	The sequence number of the entry
	Date and time	29	The date and time when the entry was logged
	Source program	32	The name of the programin which the event occurred
	Source process ID	10	The ID of the process in which the event occurred
	Source location	255	Information identifying the host where the event occurred
	Subject ID information	256	Information identifying the user who caused the event
	Object information	256	The service name Output only when an entry is logged from within a service function; not output otherwise.
	Object location information	64	The user server name

Specified by	Item	Output length (max. bytes)	Description
	Request source host	255	Information identifying the host that sent the request, when the event involves the linking of multiple programs Not output if there is no information about the request source host.
	Location ID information	64	The path specified in the `DCDIR` environment variable

2.7 User journal acquisition

Any information from UAPs can be output to system journal files as user journals (UJs). A user journal can be acquired by using the function `dc_jnl_ujput()` [`CBLDCJNL('UJPUT  ')`].

A user journal acquisition facility can be used only with TP1/Server Base. No user journal can be acquired by a UAP with TP1/LiNK.

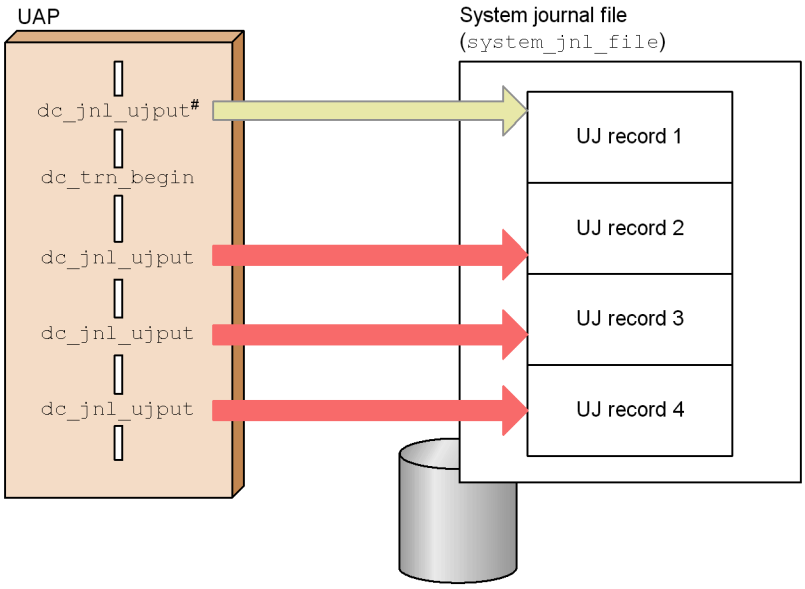
Units in which user journals are acquired by using the function `dc_jnl_ujput()` are called *UJ records*. When the function `dc_jnl_ujput()` is called once, one UJ record is acquired.

You can acquire a UJ record either outside or inside the range of a transaction. A UJ record that is acquired outside the range of a transaction is called a *UJ from outside the transaction*. A UJ record that is acquired inside the range of a transaction is called a *UJ from inside the transaction*. A UJ record that is outside the transaction is output to the system journal file when the journal buffer becomes full or when a transaction of another application terminates normally (when the transaction processing is committed).

To acquire the UJ record using an application that does not generate transactions, call the function `dc_jnl_ujput()` in which `DCJNL_FLUSH` is set for `flags` at the appropriate timing.

The figure below shows acquisition of user journals.

Figure 2-53: Acquiring user journals



#: A UJ record from outside the transaction is output to the system journal file when the journal buffer becomes full, or at the synchronization point (when the tranaction is committed) when the transaction of another application terminates normally.

If an error occurs in the transaction that called the function `dc_jnl_ujput()`, user journal acquisition processing cannot be invalidated by executing rollback processing. Even if the UAP process that called this function is recovered partially, the UJ record is output to the system journal file.

2.8 Journal data editing

A file to which the execution result of the `jnlrput` command is redirected can be edited from a UAP by using a function. Only an API in COBOL language can support journal data editing. There is no API in C language for this facility.

To call the function from a UAP:

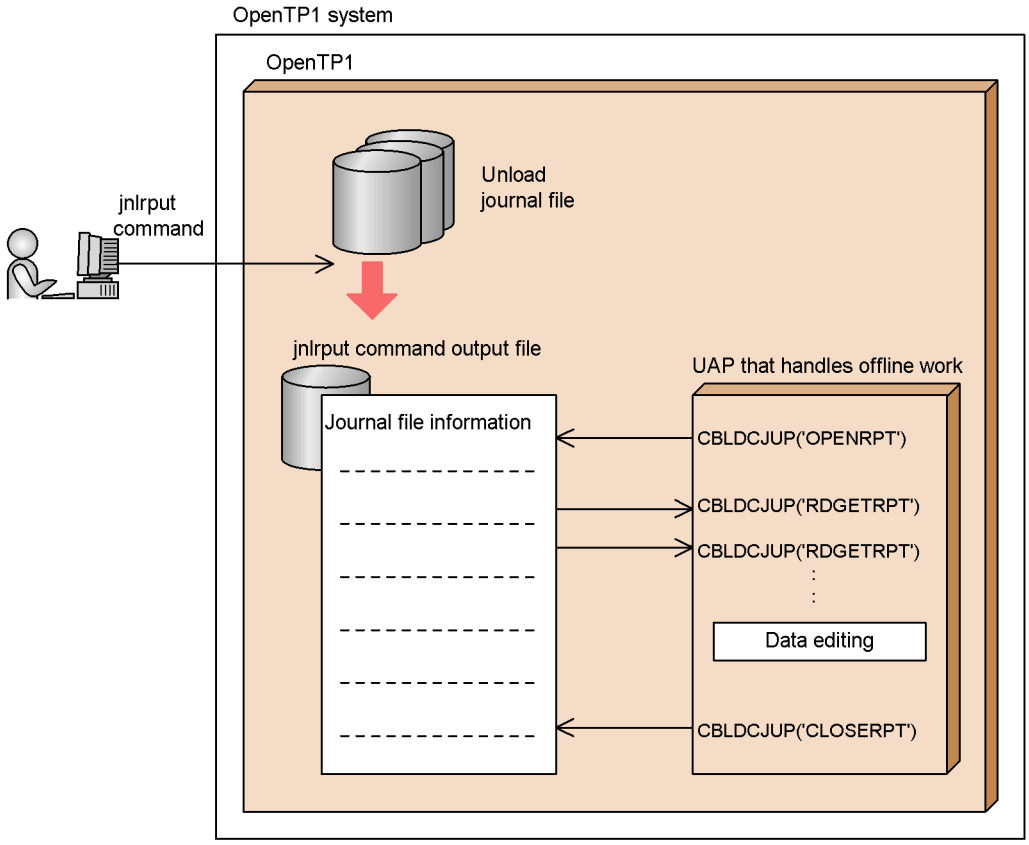
1. Open the `jnlrput` output file with `CBLDCJUP ('OPENRPT')`.
2. Enter journal data with `CBLDCJUP ('RDGETRPT')`. Enter journal data one for each journal data type. Call `CBLDCJUP ('RDGETRPT')` repeatedly until required journal data is entered completely.
3. Edit data in UAP processing.
4. Close the `jnlrput` output result file with `CBLDCJUP ('CLOSERPT')`.

Only the UAP that handles offline work can access the output file of the `jnlrput` command. Other UAPs are not permitted to access the jnlrput output file.

The journal data editing facility can be used only with TP1/Server Base. APIs for journal data editing cannot be used with TP1/LiNK.

The figure below shows journal data editing.

Figure 2-54: Journal data editing



2.9 Receiving message log notification

Reception-dedicated application programs created in the system can be notified of OpenTP1 message logs. The application program, upon receiving a notification, can notify other vendors' network management system of the OpenTP1 status.

To enable message log notification, assign `Y` to the `log_notify_out` operand in the OpenTP1 log service definition.

(1) Application programs that can receive message log notification

Only application programs created for reception can receive message log notification. OpenTP1 UAPs (SUPs, SPPs, and MHPs) cannot receive a message log notification.

When receiving a notification, the application program uses OpenTP1 functions. When writing an application program, include the header file of the OpenTP1 log service and link the OpenTP1 library.

For an application program to receive a notification, the environment variable `DCDIR` that identifies the OpenTP1 home directory must be set. The value assigned to this environment variable must be the same as for the OpenTP1 that will send the message log notification.

If you need all message logs generated since OpenTP1 online operation started, the application program to receive notifications must be started before the OpenTP1.

(2) Procedure for receiving a message log notification

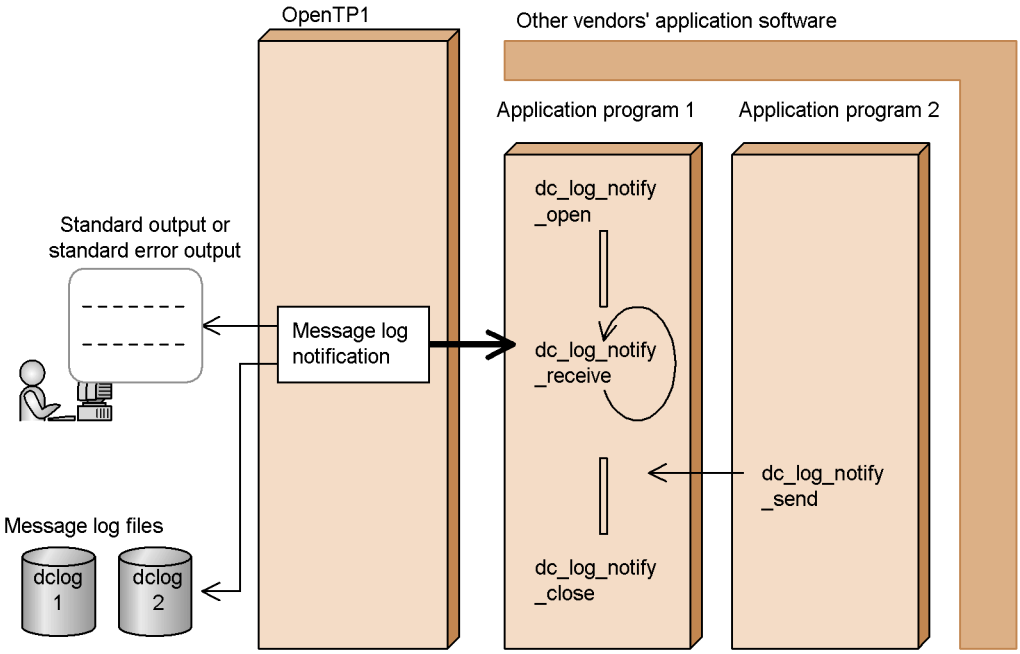
The application program to receive notifications must declare the starting of reception using the function `dc_log_notify_open()`. It receives message logs using the function `dc_log_notify_receive()`. Only one message log can be received by one run of the function `dc_log_notify_receive()`. To receive multiple message logs, repeatedly invoke the function `dc_log_notify_receive()`.

To terminate receiving message log notifications, invoke the function `dc_log_notify_close()`. Even after invoking the function `dc_log_notify_close()`, you can restart receiving message log notifications by invoking the function `dc_log_notify_open()`.

Even after the OpenTP1 terminates, the notification reception application program continues to wait until the function `dc_log_notify_close()` is invoked. To notify a waiting application program of reception completion, send data from another application program using the function `dc_log_notify_send()`. The application program for reception completion notification cannot invoke the function `dc_log_notify_open()` before invoking the function `dc_log_notify_send()`.

The figure below shows the reception of a message log notification.

Figure 2-55: Reception of message log notification



(3) Notes on receiving message log notifications

Notes on receiving message log notifications are given below.

- The functions `dc_log_notify_open()`, `dc_log_notify_receive()`, `dc_log_notify_close()`, and `dc_log_notify_send()` cannot be executed from within an interrupt routine.
- Message log notifications may or may not be received from the OpenTP1 depending on the time the function `dc_log_notify_receive()` is invoked. The following message logs cannot be received:

  1. Message logs output by the OpenTP1 when the application program is inactive, before the application program invokes the function `dc_log_notify_open()`, or after the application program invokes the function `dc_log_notify_close()`
  2. Message logs that come after the message log receive buffer has run short of space because the function `dc_log_notify_receive()` was not invoked when a previous message log was reported by the OpenTP1

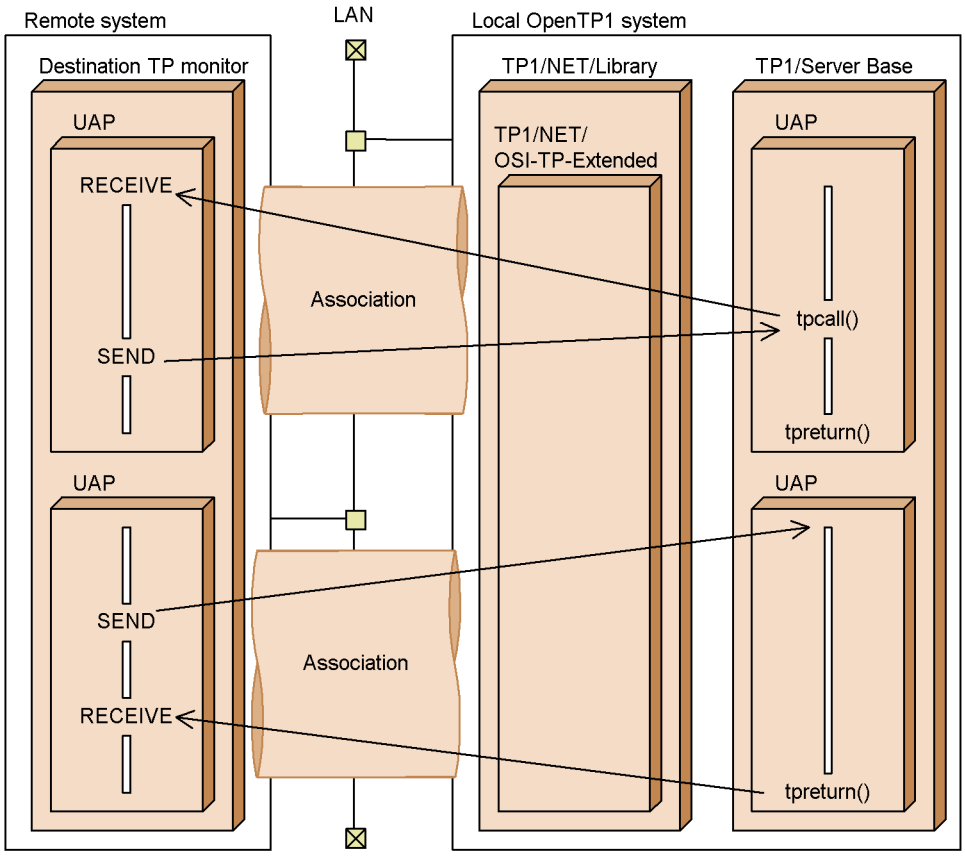
2.10 Client/server mode communication using OSI TP

TCP/IP and OSI TP can be used as communication protocols for OpenTP1 client/server mode communication. This section outlines communication using OSI TP as the communication protocol. This communication requires TP1/NET/Library, TP1/NET/OSI-TP-Extended, and products for managing communication under OSI TP. In addition, an OpenTP1 system service (XATMI communication service) is necessary.

Client/server mode communication using OSI TP as the communication protocol is possible only when the basic facility of the OpenTP1 is the TP1/Server Base. If the TP1/LiNK is used, OSI TP communication is impossible.

The figure below shows the concept of client/server mode communication using OSI TP.

Figure 2-56: Concept of client/server mode communication using OSI TP



2.10.1 Application programs used for OSI TP communication

OpenTP1 UAPs use the XATMI interface for communication with remote systems. SUPs and SPPs are OpenTP1 UAPs that can be used for client/server mode communication using OSI TP. Other OpenTP1 UAPs (MHPs) cannot be used.

UAPs need not be aware of the protocol for internode communication.

(1) Relationship with transaction processing

An OpenTP1 system can extend transaction processing to a remote OpenTP1 system. When an OpenTP1 system is communicating with a non-OpenTP1 system, it can extend transaction processing to the remote system using OSI TP.

2.10.2 SPPs for a communication event

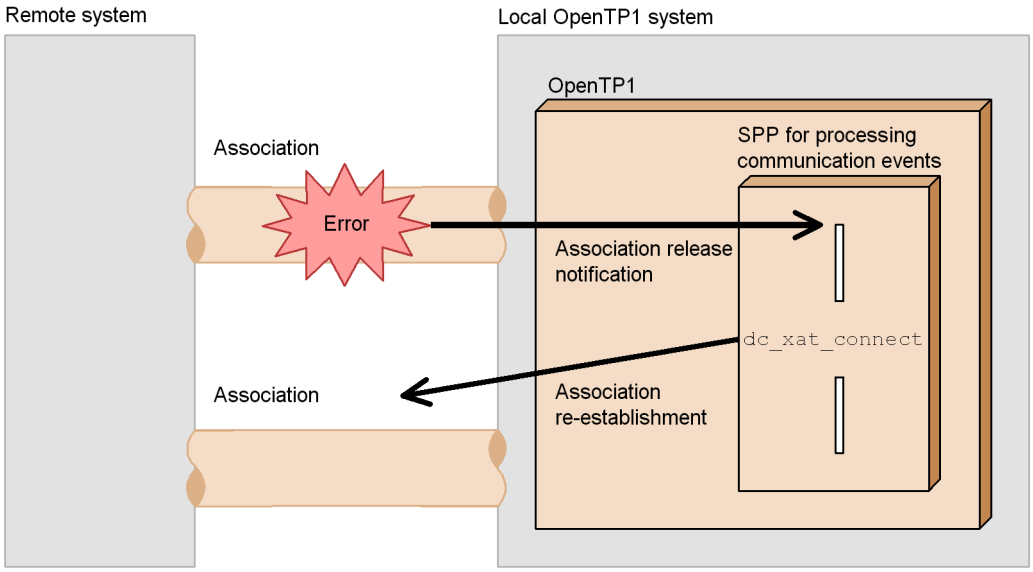
For client/server mode communication using OSI TP, it is necessary to create an SPP that obtains information about the establishment and release of associations. This SPP is referred to as an *SPP for a communication event*. Once you create a SPP for a communication event, you can receive a communication event notifying you of association release due to an error. By receiving this communication event, you can know when to re-establish the association. In addition, SPP for a communication event can obtain the attribute and status of the association from detailed information contained in the communication event it has received.

When an association is established or released, the XATMI communication service starts the SPP for a communication event by requesting server with a nonresponse RPC. Communication events are reported regardless of whether the local system is the initiating or recipient system.

For details on information received by the SPP for a communication event, see the applicable *OpenTP1 Programming Reference* manual.

The figure below outlines the SPP for a communication event.

Figure 2-57: Outline of SPP for a communication event



(1) System definitions related to SPP for a communication event

Before a SPP for a communication event can receive communication events, its service group name and service name must be specified in the XATMI communication service definition. Communication events that can be received by the SPP vary depending on the operand to which the service group name and service name are assigned as follows:

`xat_aso_con_event_svcname` operand:

Communication events reporting association establishment

`xat_aso_discon_event_svcname` operand:

Communication events reporting normal association release

`xat_aso_failure_event_svcname` operand:

Communication events reporting abnormal association release

If you assign the same service group name and service name to multiple operands, the SPP for a communication event can receive multiple types of communication events.

Assign `betran` to the `server_type` operand in the user service definition for the SPP for a communication event.

(2) Association establishment for SPP for a communication event

The SPP for a communication event can invoke a function to establish an association. The function `dc_xat_connect()` [`CBLDCXAT('CONNECT')`] is used for this

purpose. When this function returns, the SPP for a communication event can receive information about the fact that an association has been normally established.

The function `dc_xat_connect()` can establish an association only if the local system is the initiating side. In addition, since the function return is not synchronized with the association establishment, the service function that has invoked the function `dc_xat_connect()` cannot receive the communication event that reports the association establishment.

(3) Conditions for reporting association status

Association establishment is reported in the following cases:

- Association establishment at the time of OpenTP1 system start
- Association establishment caused by `nettactcn` command execution
- Association establishment requested by SPP for a communication event
- Association establishment initiated by remote system

Association release is reported in the following cases:

- Forced association release caused by `nettactcn` command execution
- Association release caused by error in lower layer
- Association release caused by fault in TP1/NET/OSI-TP-Extended
- Association release caused by XATMI communication service failure
- Failure in association release
- Normal association release initiated by remote system
- Forced association release initiated by remote system

2.10.3 Errors encountered during OSI TP communication

When an error occurs during client/server mode communication using OSI TP, the XATMI interface function that has requested the service returns with an error. For details on the values that may be returned, see the notes on the pertinent XATMI interface function in the applicable *OpenTP1 Programming Reference* manual.

When a communication protocol error occurs, take action according to the troubleshooting procedure in the manual *OpenTP1 Protocol TP1/NET/ OSI-TP-Extended*.

2.11 Acquiring performance verification traces

Trace information is acquired for all main events that occur in each service running on OpenTP1. This process is called a *performance verification trace* (`prf` *trace*). A performance verification trace is made up of trace information intended for enhancing the efficiency of performance verifications and troubleshooting. A performance verification trace has the following features:

- You can acquire a trace even if the information extends over nodes or processes.
- You can acquire traces in units of internal events instead of units of APIs. This enables you to determine which area of processing is hindering performance.

TP1/Extension 1 must be installed before you can use this facility. Note that operation will be unpredictable if you run the facility while TP1/Extension 1 is not installed.

To acquire a user-specific performance verification trace from a UAP, call the function `dc_prf_utrace_put()` [`CBLDCPRF('PRFPUT  ')`].

To find out the acquired sequential trace number of the latest performance verification trace within the process, call the function `dc_prf_get_trace_num()` [`CBLDCPRF('PRFGETN ')`]. The acquired sequential trace number of the latest performance verification trace within the process is reported to the call source of the function `dc_prf_get_trace_num()`.

2.12 Real-time statistical information acquisition

The execution time and the execution count in an arbitrary section in the UAP can be acquired as real-time statistical information. Note that real-time statistical information for arbitrary sections cannot be acquired if the target UAP performs offline processing.

To acquire real-time statistical information in an arbitrary section, call the function `dc_rts_utrace_put()` [`CBLDCRTS('RTSPUT  ')`] from the UAP.

Specify the item to be acquired in `event_id` and the action related to acquisition in `flags`. The table below lists the actions that can be specified with `flags`.

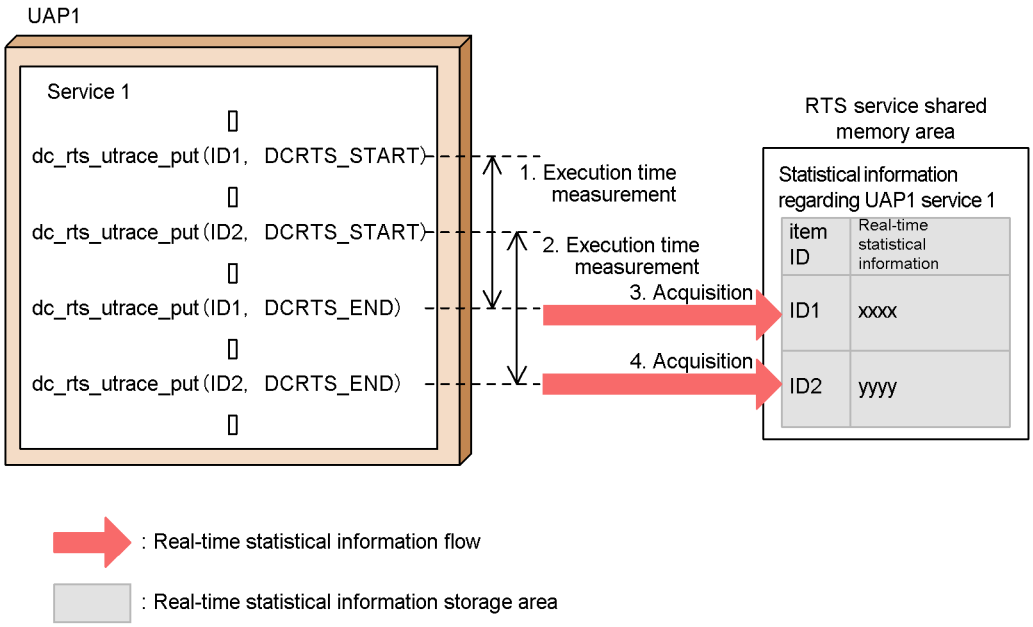
Table 2-4: Specifying the flags argument to the function dc_rts_utrace_put()

flags value	Acquisition-related action
`DCRTS_START`	Start execution time measurement.
`DCRTS_END`	Acquire the execution time and end measurement.
`DCNOFLAGS`	Acquire only the execution count.

The execution count and the execution time acquired by the function `dc_rts_utrace_put()` are edited and output as real-time statistical information for the item ID assigned to `event_id`.

The figure below provides an example of acquiring real-time statistical information for arbitrary sections.

Figure 2-58: Example of acquiring real-time statistical information in arbitrary sections



1. Start execution time measurement for item ID 1.
2. Start execution time measurement for item ID 2.
3. End execution time measurement for item ID 1 and acquire statistical information (the execution time and the execution count) in the RTS service shared memory area.
4. End execution time measurement for item ID 2 and acquire statistical information (the execution time and the execution count) in the RTS service shared memory area.

Chapter

3. Facilities Provided by TP1/ Message Control

This chapter explains the facilities which are available to application programs that use the message exchange facility (TP1/Message Control).

The facilities are explained using C-language function names. For each function, the name of the equivalent COBOL-language API function is indicated in brackets [ ] when the function appears first in this chapter. After that, only the C-language function name is written.

This chapter contains the following sections:

3.1 MCF communication service operations
3.2 Connection establishment and release
3.3 Application-related operations
3.4 Shutdown and release of logical terminals
3.5 Communication protocol products and functions available in operations
3.6 Message exchange processing
3.7 MCF transaction control
3.8 MCF extended facilities
3.9 User exit routines
3.10 MCF events
3.11 MCF processes used by application programs

3.1 MCF communication service operations

This section explains MCF communication service operations. For details on MCF communication service operations that use operation commands, see the manual *OpenTP1 Operation*.

(1) Displaying the status of MCF communication services

The status of MCF communication services and the application start process can be displayed using the function `dc_mcf_tlscom()`[`CBLDCMCF('TLSCOM  ')`]. Information such as the MCF communication server name, the MCF communication server process ID, and the MCF communication service status can be displayed.

(2) Functional differences between the API and the operation command (MCF communication service operations)

The table below shows the functional differences between the function and the operation command used for MCF communication service operations.

Table 3-1: Functional differences between the function and the operation command (MCF communication service operations)

Function name	Operation command name	Functional differences
`dc_mcf_tlscom`	`mcftlscom`	1. Acquires the status of all MCF communication services. Cannot acquire the status of specific MCF communication services only. 2. Cannot acquire MCF communication service process IDs.

3.2 Connection establishment and release

To use TP1/Messaging Control to exchange messages with the mainframe or a workstation, a logical communication path (connection) is established between the local system and the remote system.

This section explains how to issue a function from the UAP to establish or release a connection. For details on using operation commands to establish or release a connection, see the manual *OpenTP1 Operation*.

3.2.1 Establishing or releasing a connection by issuing a function from the UAP

The function `dc_mcf_tactcn()` [`CBLDCMCF('TACTCN  ')`] is used to establish a connection, and the function `dc_mcf_tdctcn()` [`CBLDCMCF('TDCTCN  ')`] is used to release a connection. Furthermore, the function `dc_mcf_tlscn()` [`CBLDCMCF('TLSCN   ')`] can be used to acquire the connection status.

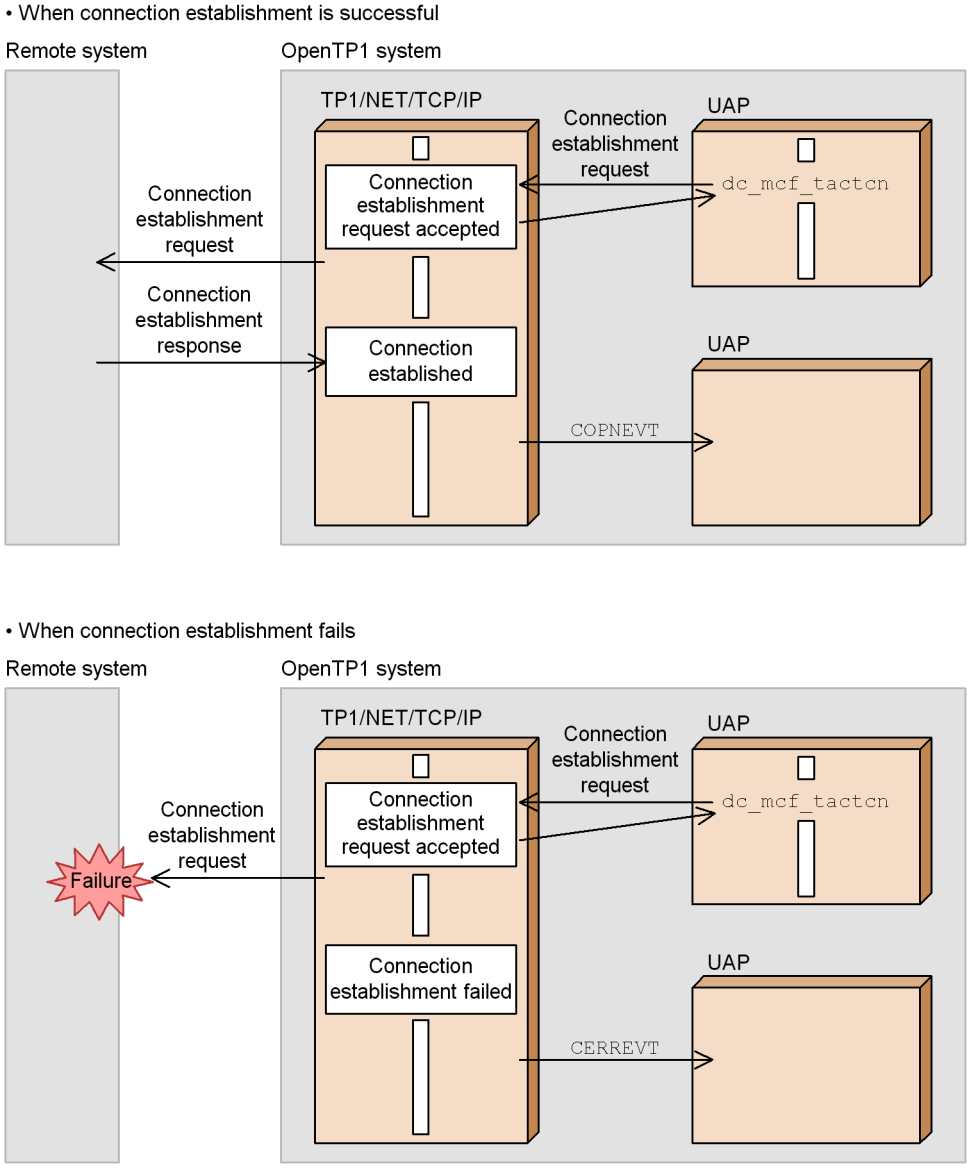
In TP1/NET/UDP, connection management functions cannot be used.

(1) Establishing a connection

Issuing the function `dc_mcf_tactcn()` requests the MCF communication process to establish a connection with the remote system.

Depending on the protocol used, an MCF event is reported to the UAP when a connection to the remote system is established or when connection establishment fails. The figure below shows the flow for establishing a connection, using an example in which the function `dc_mcf_tactcn()` is used on TP1/NET/TCP/IP.

Figure 3-1: Example of establishing a connection using the function `dc_mcf_tactcn()`

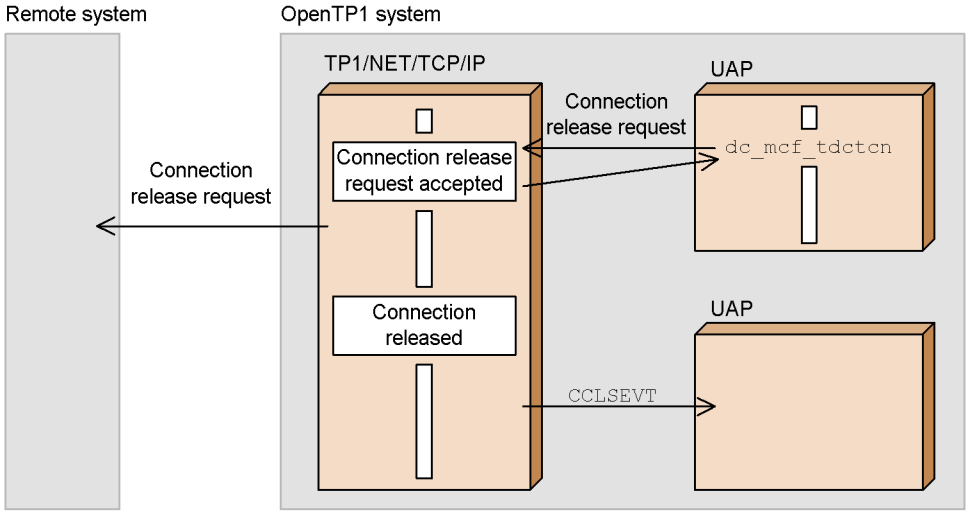


(2) Releasing a connection

Issuing the function `dc_mcf_tdctcn()` requests the MCF communication process to release the connection with the remote system.

Depending on the protocol used, an MCF event is reported to the UAP when the connection is released. The figure below shows the flow for releasing a connection, using an example in which the function dc_mcf_tdctcn() is used on TP1/NET/TCP/IP.

Figure 3-2: Example of releasing a connection using the function dc_mcf_tdctcn()



(3) Functional differences between APIs and operation commands (connection establishment and release)

The table below shows the functional differences between the functions and the operation commands used for establishing or releasing a connection.

Table 3-2: Functional differences between functions and operation commands (connection establishment and release)

Function name	Operation command	Functional differences
dc_mcf_tactcn	mcftactcn	1. Requests establishment of a single connection. Multiple or batch specification of connections is not allowed. 2. A request to establish a connection group is not allowed. 3. Subconnections cannot be specified. 4. A connected XP service cannot be specified.
dc_mcf_tdctcn	mcftdctcn	1. Requests release of a single connection. Multiple or batch specification of connections is not allowed. 2. A request to release a connection group is not allowed. 3. Subconnections cannot be specified.

Function name	Operation command	Functional differences
dc_mcf_tlscn	mcftlscn	1. Acquires the status of a single connection. Multiple or batch specification of connections is not allowed. 2. The status of a connection group cannot be acquired. 3. Only the protocol type and connection status can be acquired. Other additional information or protocol-specific information cannot be acquired.

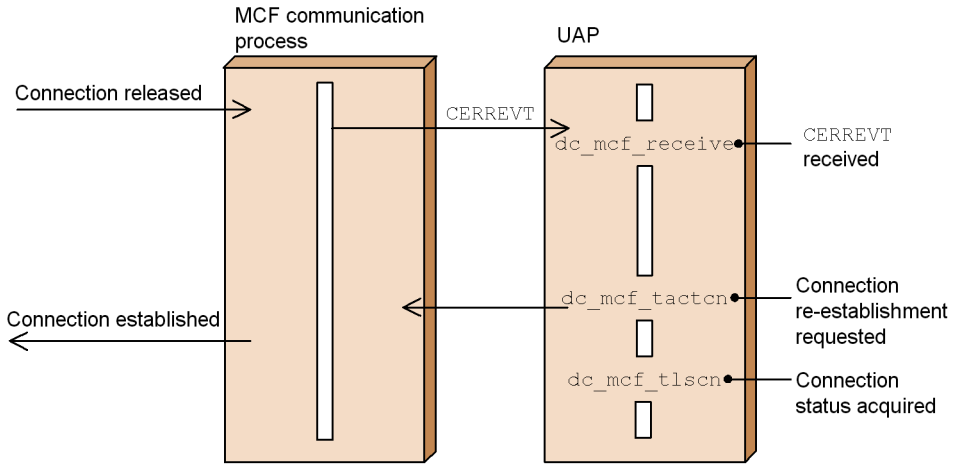
3.2.2 Coding examples for re-establishing or forcibly releasing a connection

This subsection provides coding examples for re-establishing or forcibly releasing a connection.

(1) Coding example for re-establishing a connection

The figure and coding example below show an example of automatically re-establishing a connection after CERREVT (connection error) is reported.

Figure 3-3: UAP example for automatically re-establishing a connection




```
void cerrevt(){
    char            rcvdata[256];
    DCLONG          rcv_len;
    DCLONG          rtime;
    int             rtn;
    dcmcf_tactcnopt cnopt;
    dcmcf_tlscnopt  cnopt2;
    DCLONG  infcnt = 1;
```

```
dcmcf_cninf     inf;

rtn = dc_mcf_receive(DCMCFFRST, DCNOFLAGS, termnam, "",
                     rcvdata,&rcv_len, sizeof(rcvdata),
                     &rtime);
if (DCMCFRTN_00000 == rtn){

/* Processing during connection release          */
/*                     :                         */

/* Connection re-establishment request           */
    memset(&cnopt, 0, sizeof(cnopt));
    strcpy(cnopt.idnam, termnam);

    rtn = dc_mcf_tactcn(DCMCFLE, &cnopt, NULL, NULL,
                        NULL, NULL);
    if (DCMCFRTN_00000 == rtn){
/* Acceptance of connection                      */
/* re-establishment request: Successful          */

        while(1){
        /* Connection status acquisition */
            memset(&cnopt2, 0, sizeof(cnopt2);
            strcpy(cnopt2.idnam, termnam);
            memset(&inf, 0, sizeof(inf));

            rtn = dc_mcf_tlscn(DCMCFLE, &cnopt2, NULL,
                               NULL, NULL, &infcnt,
                               &inf, NULL);
            if (DCMCFRTN_00000 == rtn){
                if (DCMCF_CNST_ACT == inf.status){
                    /* Connection established */
                    break;
                }
            } else {
                /* Error processing */
            }
            sleep(1);
        }

/* Processing following connection establishment */
/*                     :                         */

    } else {
/* Acceptance of connection                      */
/* re-establishment request: Failed              */
        /* Error processing */
    }
```

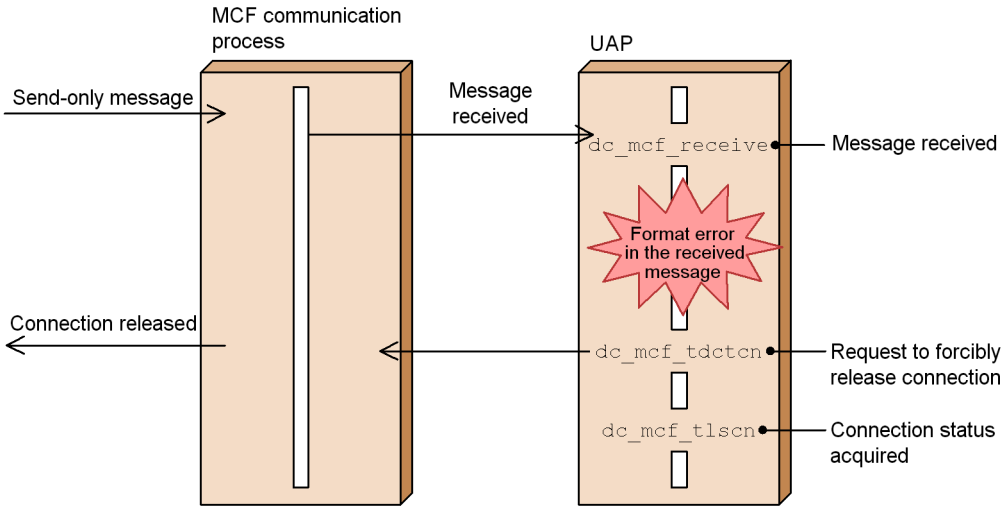
```
    } else {
        /* Error processing */
    }

    return;
}
```

(2) Coding example for forcibly releasing a connection

The figure and coding example below show an example of forcibly releasing a connection when the received message contains a format error.

Figure 3-4: UAP example for forcibly releasing a connection




```
void mhprecv(){
    char            rcvdata[256];
    DCLONG          rcv_len;
    DCLONG          rtime;
    int             rtn;
    int             check;
    dcmcf_tdctcnopt cnopt;
    dcmcf_tlscnopt  cnopt2;
    DCLONG  infcnt = 1;
    dcmcf_cninf     inf;

    rtn = dc_mcf_receive(DCMCFFRST, DCNOFLAGS, termnam, "",
                         rcvdata, &rcv_len, sizeof(rcvdata),
                         &rtime);
    if (DCMCFRTN_00000 == rtn){
```

```
    /* Checking of the received message */
    /*            :                */

    if (0 == check){
        /* Checking result: Valid */
        /* Processing when the result is normal */
    } else {
        /* Checking result: Invalid */

        /* Request to forcibly release the connection */
        memset(&cnopt, 0, sizeof(cnopt));
        strcpy(cnopt.idnam, termnam);

        rtn = dc_mcf_tdctcn(DCMCFLE | DCMCFFRC, &cnopt,
                            NULL, NULL, NULL, NULL);
        if (DCMCFRTN_00000 == rtn){
/* Acceptance of forcible                          */
/* connection release request: Successful         */

            while(1){
                /* Connection status acquisition */
                memset(&cnopt2, 0, sizeof(cnopt2);
                strcpy(cnopt2.idnam, termnam);
                memset(&inf, 0, sizeof(inf));

                rtn = dc_mcf_tlscn(DCMCFLE, &cnopt2, NULL,
                                   NULL, NULL, &infcnt,
                                   &inf, NULL);
                if (DCMCFRTN_00000 == rtn){
                    if (DCMCF_CNST_DCT == inf.status){
                        /* Connection released */
                        break;
                    }
                } else {
                    /* Error processing */
                }
                sleep(1);
            }

/* Processing following connection release        */
/*                    :                           */

        } else {
 /* Acceptance of forcible                 */
 /* connection release request: Failed     */
         /* Error processing */
        }
```

```
            }

        } else {
            /* Error processing */
        }

        return;
    }
```

3.2.3 Start and terminate acceptance of connection establishment requests

The function `dc_mcf_tonln()` [`CBLDCMCF('TONLN    ')`] is used to start acceptance of connection establishment requests, and the function `dc_mcf_tofln()` [`CBLDCMCF('TOFLN    ')`] is used to terminate acceptance of connection establishment requests. Furthermore, the function `dc_mcf_tlsln()` [`CBLDCMCF('TLSLN    ')`] can be used to acquire the establishment request acceptance status.

For details, see the applicable *OpenTP1 Protocol* manual.

(1) Functional differences between APIs and operation commands (start and terminate acceptance of connection establishment requests)

The table below shows the functional differences between the functions and the operation commands used to start or terminate acceptance of connection establishment requests.

Table 3-3: Functional differences between functions and operation commands (start and terminate acceptance of connection establishment requests)

Function name	Operation command	Functional differences
`dc_mcf_tlsln`	`mcftlsln`	1. The MCF communication process identifier of the target MCF communication process must be specified. Furthermore, the acceptance status of the server-type connection establishment requests of all MCF communication processes cannot be acquired. 2. Only the acceptance status of the server-type connection establishment requests can be acquired. Other additional information cannot be acquired.
`dc_mcf_tofln`	`mcftofln`	None
`dc_mcf_tonln`	`mcftonln`	None

3.3 Application-related operations

This section explains application-related operations. For details on using operation commands for application-related operations, see the manual *OpenTP1 Operation*.

(1) Deleting application timer start requests

The function `dc_mcf_adltap()` [`CBLDCMCF('ADLTAP  ')`] is used to stop the start of an application that requested a timer start. By issuing the function `dc_mcf_adltap()`, you can delete the timer start request for the specified application and stop the start of that application.

(2) Functional differences between the API and the operation command (application-related operations)

The table below shows the functional differences between the function and the operation command used for application-related operations.

Table 3-4: Functional differences between the function and operation command (application-related operations)

Function name	Operation command	Functional differences
`dc_mcf_adltap`	`mcfadltap`	1. Deletes a single application timer start request. Multiple or batch specification of applications is not allowed. 2. The application start process identifier of the application start process must be specified. Furthermore, the timer start requests of all application start processes cannot be deleted.

3.4 Shutdown and release of logical terminals

This section explains how to issue functions from the UAP to shut down or release logical terminals. For details on how to use operation commands to shut down or release logical terminals, see the manual *OpenTP1 Operation*.

(1) Displaying the status of a logical terminal

The function `dc_mcf_tlsle()` [`CBLDCMCF('TLSLE  ')`] can be used to display the status of a logical terminal. Information such as the MCF identifier, the logical terminal name, and the logical terminal status (regardless of whether the terminal is shut down) can be displayed.

The logical terminal status is stored in the area specified inside the UAP.

(2) Shutting down or releasing a logical terminal

When a logical terminal is shut down, it cannot send messages requested by the UAP to the remote system. In this state, if the UAP makes a message transmission request, it is accepted normally, but the message to be transmitted remains in the output queue. Furthermore, in this state, scheduling of messages received from the remote system is done normally.

The function `dc_mcf_tdctle()` [`CBLDCMCF('TDCTLE  ')`] is used to shut down a logical terminal. While the logical terminal is shut down, requests to send send-only messages remain in the output queue. Note that a logical terminal might also be shut down by an error.

On the other hand, when a logical terminal is released, its functions can be used.

The function `dc_mcf_tactle()` [`CBLDCMCF('TACTLE  ')`] is used to release a logical terminal. When it is released, the messages remaining in the output queue are sent. Note that the logical terminal cannot be released if no connection has been established.

(3) Deleting the content of the output queue of a logical terminal

The function `dc_mcf_tdlqle()` [`CBLDCMCF('TDLQLE  ')`] is used to discard the messages remaining in the output queue after a connection is established.

Issuing the function `dc_mcf_tdlqle()` deletes all messages remaining in the output queues of the disk queue and the memory queue, and starts an MCF event for each deleted message. Before the function `dc_mcf_tdlqle()` can be issued, the logical terminal must be shut down using the `mcftdctle` command or the function `dc_mcf_tdctle()`.

(4) Functional differences between APIs and operation commands (shutdown and release of logical terminals)

The table below shows the functional differences between functions and operation commands used to shut down or release a logical terminal.

Table 3-5: Functional differences between functions and operation commands (shutdown and release of logical terminals)

Function name	Operation command	Functional differences
dc_mcf_tactle	mcftactle	1. Requests the release of a single logical terminal. Multiple or batch specification of logical terminals is not allowed. 2. Releases both the logical terminal and its queue. Specification of only one or the other is not allowed.
dc_mcf_tdctle	mcftdctle	1. Requests the shutdown of a single logical terminal. Multiple or batch specification of logical terminals is not allowed. 2. Shuts down both the logical terminal and its queue. Specification of only one or the other is not allowed.
dc_mcf_tdlqle	mcftdlqle	1. Requests the deletion of the output queue of a single logical terminal. Multiple or batch specification of logical terminals is not allowed. 2. Deletes both the disk queue and the memory queue. Specification of only one or the other is not allowed. 3. If the MCF event (ERREVTA) that reports discarding of an unsent message is defined in the MCF application definition, ERREVTA is reported. The reporting cannot be suppressed.
dc_mcf_tlsle	mcftlsle	1. Acquires the status of a single logical terminal. Multiple or batch specification of applications is not allowed. 2. Only the status of the logical terminal can be acquired. Other additional information cannot be acquired.

3.5 Communication protocol products and functions available in operations

This section explains which functions are available in each of the operations used by OpenTP1-provided products conforming to different communication protocols. Here, *operations* refers to the following:

- MCF communication service operations
- Connection establishment and release
- Application-related operations
- Shutdown and release of logical terminals

The following tables show which functions are available in each of the operations used by OpenTP1-provided products conforming to different communication protocols.

Table 3-6: Communication protocol products and functions available in operations (1/3)

Function name	Communication protocol product			
	TP1/NET/HDLC	TP1/NET/HSC	TP1/NET/NCSB	TP1/NET/ OSAS-NIF
dc_mcf_adltap	Y	Y	Y	Y
dc_mcf_tactcn	Y	Y	Y	Y
dc_mcf_tactle	Y	Y	Y	Y
dc_mcf_tdctcn	Y	Y	Y	Y
dc_mcf_tdctle	Y	Y	Y	Y
dc_mcf_tdlqle	Y	Y	Y	Y
dc_mcf_tlscn	Y	Y	Y	Y
dc_mcf_tlscom	Y	Y	Y	Y
dc_mcf_tlsle	Y	Y	Y	Y
dc_mcf_tlsln	N	N	N	N
dc_mcf_tofln	N	N	N	N
dc_mcf_tonln	N	N	N	N

Legend:

Y: Can be used.

N: Cannot be used.

Table 3-7: Communication protocol products and functions available in operations (2/3)

Function name	Communication protocol product			
	TP1/NET/OSI-TP	TP1/NET/SLU - TypeP2	TP1/NET/TCP/IP	TP1/NET/User Agent
dc_mcf_adltap	Y	Y	Y	Y
dc_mcf_tactcn	Y	Y	Y	Y
dc_mcf_tactle	N	Y	Y	Y
dc_mcf_tdctcn	Y	Y	Y	Y
dc_mcf_tdctle	N	Y	Y	Y
dc_mcf_tdlqle	N	Y	Y	Y
dc_mcf_tlscn	Y	Y	Y	Y
dc_mcf_tlscom	Y	Y	Y	Y
dc_mcf_tlsle	N	Y	Y	Y
dc_mcf_tlsln	N	N	Y	N
dc_mcf_tofln	N	N	Y	N
dc_mcf_tonln	N	N	Y	N

Legend:

Y: Can be used.

N: Cannot be used.

Table 3-8: Communication protocol products and functions available in operations (3/3)

Function name	Communication protocol product			
	TP1/NET/UDP	TP1/NET/X25	TP1/NET/ X25-Extended	TP1/NET/XMAP3
dc_mcf_adltap	Y	Y	Y	Y

Function name	Communication protocol product			
	TP1/NET/UDP	TP1/NET/X25	TP1/NET/ X25-Extended	TP1/NET/XMAP3
dc_mcf_tactcn	N	Y	Y	Y
dc_mcf_tactle	Y	Y	Y	Y
dc_mcf_tdctcn	N	Y	Y	Y
dc_mcf_tdctle	Y	Y	Y	Y
dc_mcf_tdlqle	Y	Y	Y	Y
dc_mcf_tlscn	N	Y	Y	Y
dc_mcf_tlscom	Y	Y	Y	Y
dc_mcf_tlsle	Y	Y	Y	Y
dc_mcf_tlsln	N	N	N	N
dc_mcf_tofln	N	N	N	N
dc_mcf_tonln	N	N	N	N

Legend:

Y: Can be used.

N: Cannot be used.

3.6 Message exchange processing

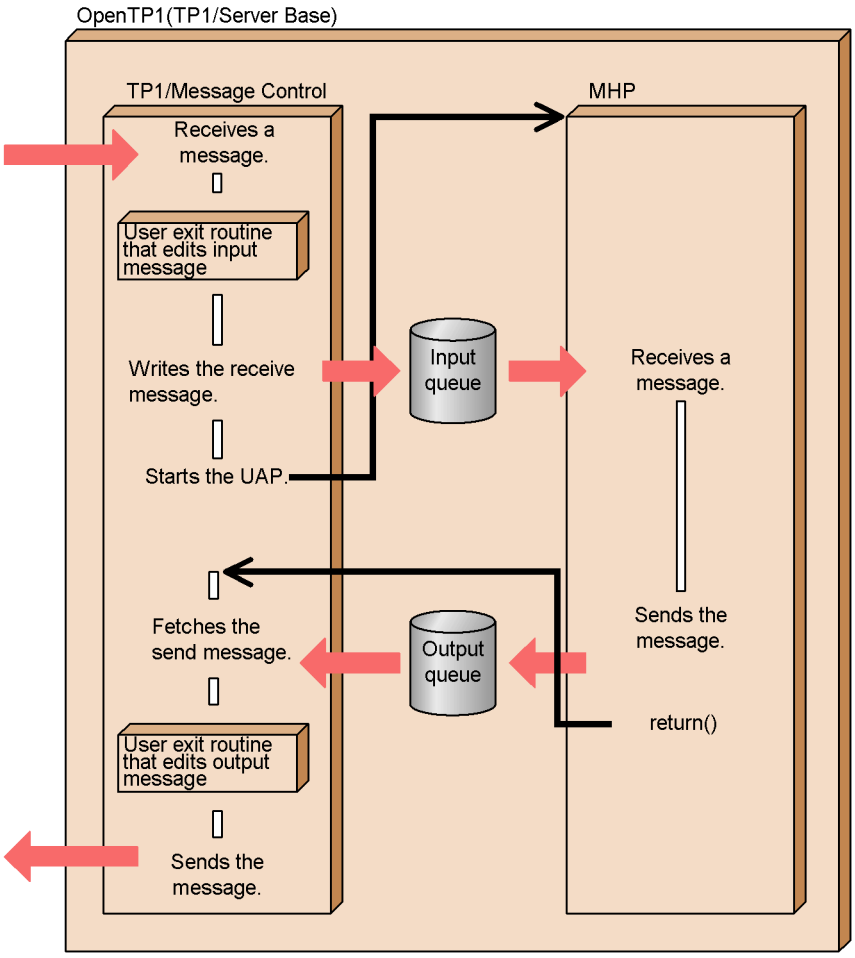
The installation of TP1/Message Control in a system furnished with the basic OpenTP1 facilities (TP1/Server Base) enables message exchange mode communication with mainframes and workstations through wide area networks (WANs), TCP/IP, and conventional networks.

MHPs are used for communication based on messages. SPPs can also be used for some message processing.

Before the message exchange facility can be available, TP1/Message Control must be installed in the system and the basic OpenTP1 facilities must be provided by TP1/Server Base. TP1/Messaging is required when you create MHPs under TP1/LiNK.

The figure below shows message exchange mode communication.

Figure 3-5: Outline of message exchange mode communication



User exit routines can be created so that UAP message processing will cover a wide variety of purposes. They can be written to meet particular requirements for jobs and environments. For details on user exit routines, see *3.9 User exit routines*.

3.6.1 Message communication modes

(1) Message communication modes available with MHPs

Message communication modes which can be used with MHPs are shown below. Available message modes vary depending on the communication protocol.

- Inquiry-response mode

  A message is received with the function `dc_mcf_receive()`

[CBLDCMCF('RECEIVE ')] from the own system, and a response message is returned with the function dc_mcf_reply() [CBLDCMCF('REPLY ')].

- Noninquiry-response mode (receive-only mode)

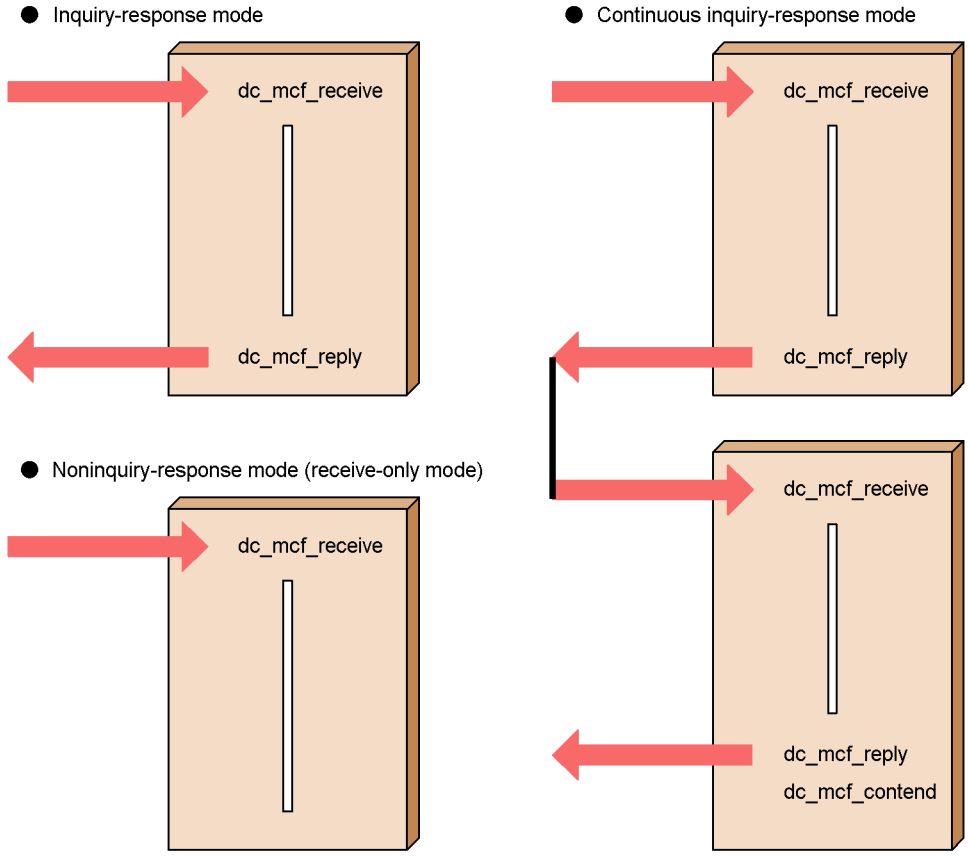
  A message is received with the function dc_mcf_receive() from the own system, but no response message is returned.

- Continuous-inquiry-response mode

  This mode is provided to continue the inquiry-response mode. A message is received with the function dc_mcf_receive() from the own system, a response message is returned with the function dc_mcf_reply(), then response processing for inquiries is continued. Use the function dc_mcf_contend() [CBLDCMCF('CONTEND ')] to terminate the continuous-inquiry-response mode.

The figure below shows the message communication modes.

Figure 3-6: Message communication modes



(2) Communication modes of MHPs and message communication facilities available to SPPs

The message communication facilities which are available to MHPs and SPPs are as follows:

- Branch send mode

  A message can be sent with the function dc_mcf_send()[CBLDCMCF('SEND')] from the another system.

- Synchronous send mode, synchronous receive mode, synchronous exchange mode

  After a message is sent to or received from the own system, the message can be sent synchronously (dc_mcf_sendsync() [CBLDCMCF ('SENDSYNC')]), received synchronously (dc_mcf_recvsync() [CBLDCMCF('RECVSYNC')]), or exchanged synchronously (dc_mcf_sendrecv() [CBLDCMCF('SENDRECV')]). The called function does not return until send processing or receive processing is completed.

To use the function dc_mcf_send() with an SPP, the SPP processing must be operating as a transaction.

(3) Message communication modes and application type

For an MHP using message exchange facilities, specify the type of application according to the message communication mode to be used. Specify the type of application for the type operand of the MCF application definition or the application attribute definition (mcfaalcap). There are the following three types of application:

- Response type (ans): MHP in inquiry-response mode
- Nonresponse type (noans): MHP in noninquiry-response mode
- Continuous-inquiry-response type (cont): MHP in continuous-inquiry-response mode

Specify noans for the mode in which a message, received with the function dc_mcf_receive(), is sent to the logical terminal of the input source by using the function dc_mcf_send().

For MHPs, specify the type of application according to the message handling mode. This specification is not required for SPPs.

If the specified type of application conflicts with the message handling mode, a message exchange function returns with an error or the MHP processing is rolled back. The type conflicts with the mode in the following cases:

- A response type MHP terminated without using the function dc_mcf_reply(). Alternatively, the MHP terminated another response type MHP which has not been activated with the function dc_mcf_execap() [CBLDCMCF('EXECAP

')].

- A nonresponse type MHP used the function `dc_mcf_reply()`.

The application type of the MCF event handling MHP is determined by the reported MCF event. See *3.10 MCF events* for details.

The table below shows the correspondence between the types of application and message exchange functions.

Table 3-9: Correspondence between the types of application and message exchange functions

Message Mode	Types of Application	Functions for Message Processing						
		receive	send	reply	send recv	recv sync	send sync	tempput, tempget contend
Inquiry-response mode	`ans`	M	Y	M	N	N	N	N
Noninquiry-response mode (receive-only mode)	`noans`	M#1	Y	N	N#2	N#2	N#2	N
Continuous-inquiry-response mode	`cont`	M	Y	M	N	N	N	Y

Legend:

M: Must be used.

Y: Can be used.

N: Cannot be used.

Note

The type of the logical terminal depends on the protocol. See the applicable *OpenTP1 Protocol* manual.

#1

The function `dc_mcf_receive()` cannot be used by SPPs.

#2

Can be called when TP1/NET/OSI-TP is used.

(4) Communication protocol products and functions available in communication modes

The following tables indicate what functions are available in each of the

communication modes used by OpenTP1-provided products conforming to different communication protocols.

Table 3-10: Functions available in communication modes used by communication protocol products (1/5)

Function name	Communication protocol product used and application type								
	TP1/NET/User Agent			TP1/NET/OSI-TP			TP1/NET/TCP/IP		
	noans type	ans type	cont type	noans type	ans type	cont type	noans type	ans type	cont type
dc_mcf_commit	Y	N	--	Y	--	--	Y	--	--
dc_mcf_receive#	Y	Y	--	Y	--	--	Y	--	--
dc_mcf_execap	Y	Y	--	N	--	--	Y	--	--
dc_mcf_reply#	N	Y	--	N	--	--	N	--	--
dc_mcf_rollback	Y	Y	--	Y	--	--	Y	--	--
dc_mcf_send#	Y	Y	--	N	--	--	Y	--	--
dc_mcf_resend#	Y	Y	--	N	--	--	Y	--	--
dc_mcf_sendrecv#	Y	Y	--	Y	--	--	Y	--	--
dc_mcf_sendsync#	N	N	--	Y	--	--	Y	--	--
dc_mcf_recvsync#	U	U	--	Y	--	--	N	--	--
dc_mcf_contend	N	N	--	N	--	--	N	--	--
dc_mcf_tempget	N	N	--	N	--	--	N	--	--
dc_mcf_tempput	N	N	--	N	--	--	N	--	--

Legend:

Y: Available with the communication protocol product

N: Unavailable

U: This communication mode is used with the communication protocol product in a unique way.

--: This communication mode cannot be used with the communication protocol product.

#

The method of using the function might vary depending on the communication protocol product. For details, see the applicable *OpenTP1 Protocol* manual.

Table 3-11: Functions available in communication modes used by communication protocol products (2/5)

Function name	Communication protocol product used and application type								
	TP1/NET/XMAP3			TP1/NET/HNA-560/20			TP1/NET/HNA-560/20 DTS		
	noans type	ans type	cont type	noans type	ans type	cont type	noans type	ans type	cont type
dc_mcf_commit	Y	N	N	Y	N	N	Y	N	N
dc_mcf_receive#	Y	Y	Y	Y	Y	Y	Y	Y	Y
dc_mcf_execap	Y	Y	Y	Y	Y	Y	Y	Y	Y
dc_mcf_reply#	N	Y	Y	N	Y	Y	N	Y	Y
dc_mcf_rollback	Y	Y	Y	Y	Y	Y	Y	Y	Y
dc_mcf_send#	Y	Y	Y	Y	Y	Y	Y	Y	Y
dc_mcf_resend#	Y	Y	Y	Y	Y	Y	Y	Y	Y
dc_mcf_sendrecv#	N	N	N	N	N	N	N	N	N
dc_mcf_sendsync#	N	N	N	N	N	N	N	N	N
dc_mcf_recvsync#	N	N	N	N	N	N	N	N	N
dc_mcf_contend	N	N	Y	N	N	Y	N	N	Y
dc_mcf_tempget	N	N	Y	N	N	Y	N	N	Y
dc_mcf_tempput	N	N	Y	N	N	Y	N	N	Y

Legend:

Y: Available with the communication protocol product

N: Unavailable

#

The method of using the function might vary depending on the communication protocol product. For details, see the applicable *OpenTP1 Protocol* manual.

Table 3-12: Functions available in communication modes used by communication protocol products (3/5)

Function name	Communication protocol product used and application type											
	TP1/NET/ OSAS-NIF			TP1/NET/ HNA-NIF			TP1/NET/ HSC(1)			TP1/NET/ HSC(2)		
	n typ	a typ	c typ	n typ	a typ	c typ	n typ	a typ	c typ	n typ	a typ	c typ
dc_mcf_commit	Y	Y	--	Y	--	--	Y	--	--	Y	--	--
dc_mcf_receive#	Y	Y	--	Y	--	--	Y	--	--	Y	--	--
dc_mcf_execap	Y	Y	--	Y	--	--	Y	--	--	N	--	--
dc_mcf_reply#	N	Y	--	N	--	--	N	--	--	N	--	--
dc_mcf_rollback	Y	Y	--	Y	--	--	Y	--	--	Y	--	--
dc_mcf_send#	Y	Y	--	Y	--	--	Y	--	--	Y	--	--
dc_mcf_resend#	Y	Y	--	Y	--	--	Y	--	--	Y	--	--
dc_mcf_sendrecv#	Y	Y	--	N	--	--	N	--	--	N	--	--
dc_mcf_sendsync#	N	N	--	N	--	--	N	--	--	Y	--	--
dc_mcf_recvsync#	U	U	--	N	--	--	N	--	--	Y	--	--
dc_mcf_contend	N	N	--	N	--	--	N	--	--	N	--	--
dc_mcf_tempget	N	N	--	N	--	--	N	--	--	N	--	--
dc_mcf_tempput	N	N	--	N	--	--	N	--	--	N	--	--

Legend:

n typ: noans type

a typ: ans type

c typ: cont type

Y: Available with the communication protocol product

N: Unavailable

U: This communication mode is used with the communication protocol product in a unique way.

--: This communication mode cannot be used with the communication protocol

product.

#

The method of using the function might vary with the communication protocol product. For details, see the applicable *OpenTP1 Protocol* manual.

Table 3-13: Functions available in communication modes used by communication protocol products (4/5)

Function name	Communication protocol product used and application type								
	TP1/NET/HDLC			TP1/NET/X25			TP1/NET/ X25-Extended		
	noans type	ans type	cont type	noans type	ans type	cont type	noans type	ans type	cont type
dc_mcf_commit	Y	--	--	Y	--	--	Y	--	--
dc_mcf_receive#	Y	--	--	Y	--	--	Y	--	--
dc_mcf_execap	Y	--	--	Y	--	--	Y	--	--
dc_mcf_reply#	N	--	--	N	--	--	N	--	--
dc_mcf_rollback	Y	--	--	Y	--	--	Y	--	--
dc_mcf_send#	Y	--	--	Y	--	--	Y	--	--
dc_mcf_resend#	Y	--	--	Y	--	--	Y	--	--
dc_mcf_sendrecv#	N	--	--	N	--	--	N	--	--
dc_mcf_sendsync#	N	--	--	N	--	--	N	--	--
dc_mcf_recvsync#	N	--	--	N	--	--	N	--	--
dc_mcf_contend	N	--	--	N	--	--	N	--	--
dc_mcf_tempget	N	--	--	N	--	--	N	--	--
dc_mcf_tempput	N	--	--	N	--	--	N	--	--

Legend:

Y: Available with the communication protocol product

N: Unavailable

--: This communication mode cannot be used with the communication protocol product.

#

The method of using the function might vary depending on the communication protocol product. For details, see the applicable *OpenTP1 Protocol* manual.

Table 3-14: Functions available in communication modes used by communication protocol products (5/5)

Function name	Communication protocol product used and application type											
	TP1/NET/SLU - TypeP1			TP1/NET/SLU - TypeP2			TP1/NET/NCSB			TP1/NET/UDP		
	n typ	a typ	c typ	n typ	a typ	c typ	n typ	a typ	c typ	n typ	a typ	c typ
dc_mcf_commit	Y	Y	--	Y	--	--	Y	--	--	Y	--	--
dc_mcf_receive#	Y	Y	--	Y	--	--	Y	--	--	Y	--	--
dc_mcf_execap	Y	Y	--	Y	--	--	Y	--	--	Y	--	--
dc_mcf_reply#	N	Y	--	N	--	--	N	--	--	N	--	--
dc_mcf_rollback	Y	Y	--	Y	--	--	Y	--	--	Y	--	--
dc_mcf_send#	Y	N	--	Y	--	--	Y	--	--	Y	--	--
dc_mcf_resend#	Y	N	--	Y	--	--	Y	--	--	Y	--	--
dc_mcf_sendrecv#	N	N	--	Y	--	--	N	--	--	N	--	--
dc_mcf_sendsync#	N	N	--	N	--	--	N	--	--	Y	--	--
dc_mcf_recvsync#	N	N	--	U	--	--	N	--	--	N	--	--
dc_mcf_contend	N	N	--	N	--	--	N	--	--	N	--	--
dc_mcf_tempget	N	N	--	N	--	--	N	--	--	N	--	--
dc_mcf_tempput	N	N	--	N	--	--	N	--	--	N	--	--

Legend:

n typ: noans type

a typ: ans type

c typ: cont type

Y: Available with the communication protocol product

N: Unavailable

U: This communication mode is used with the communication protocol product in a unique way.

--: This communication mode cannot be used with the communication protocol product.

#

The method of using the function might vary depending on the communication protocol product. For details, see the applicable *OpenTP1 Protocol* manual.

3.6.2 Message structure

This subsection explains the message structure.

(1) Logical messages and segments

A unit of data significant for inter-system communication is called a *logical message*. A logical message consists of one or more segments. A segment is a unit of information which can be processed by a single call to a library function from a UAP process.

When a logical message consists of one segment, the message can be processed by a single call to a function. When a logical message consists of multiple segments, the message should be processed by calling the same number of functions as the segments.

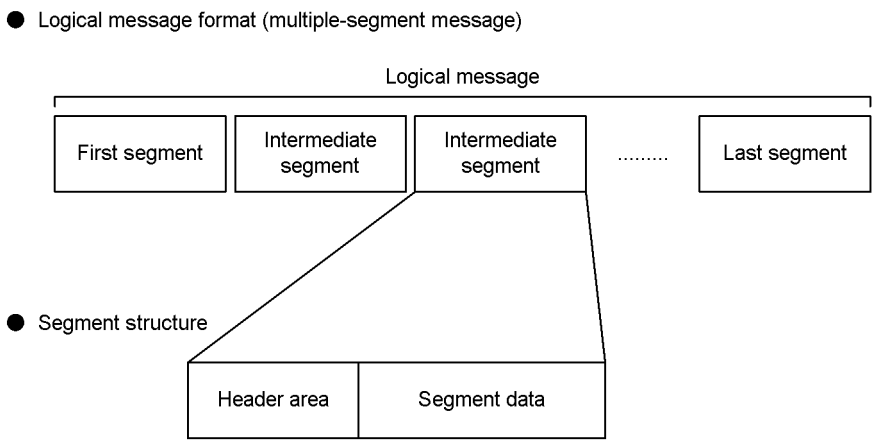
(2) Segment structure

A segment consists of the header area used by the MCF and segment data. The length of the header area determines buffer format 1 or buffer format 2. The user can decide which format should be used. However, only buffer format 2 is available with TP1/NET/XMAP3.

The length of the header area varies by the communication protocol product. For more information, see the explanation of message exchange APIs in the *OpenTP1 Protocol* manual.

The figure below shows the relationship between a logical message and segments.

Figure 3-7: Relationship between logical message and segments



3.6.3 Receiving messages

When the MCF finishes receiving the last segment of a message from another system, it passes the message to the MHP identified by the application name. The MHP calls the function `dc_mcf_receive()` [`CBLDCMCF('RECEIVE ')`] to receive the message and starts processing. This message reception is called *asynchronous message reception*.

The function `dc_mcf_receive()` receives one segment of a message at one time.

If the message consists of one segment (single-segment message), the function `dc_mcf_receive()` is called only once.

If the message consists of more than one segment, the function `dc_mcf_receive()` is called as many times as the segments. The MHP receives the message, beginning with the first segment and proceeding to intermediate segments. After receiving intermediate segments, the MCF finally receives a return value indicating that there is no more segment to be received. It then recognizes that it has received the entire message, including the last segment.

User exit routines can be used to edit messages to be passed to the MHP and to change the application name.

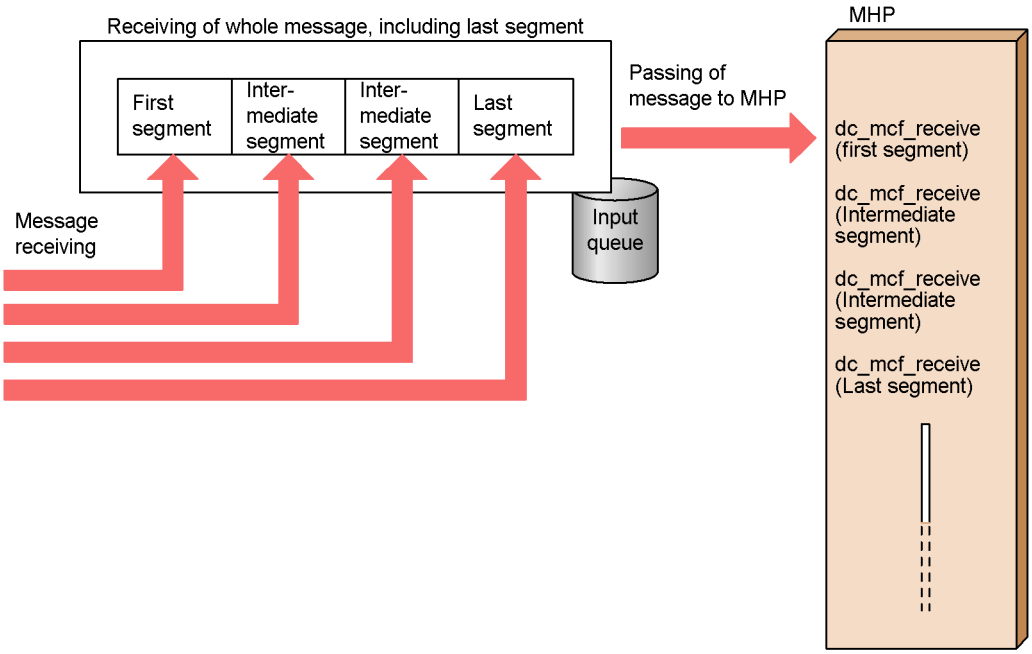
Application name:

An application name is expressed with alphanumeric characters comprising 1 to 8 bytes (from the beginning of the message to the byte followed by a space). If there is no space up to the ninth byte or the application name begins with a space, the specified application name is treated as invalid.

Application names can be edited by using the user exit routine that edits input message.

The figure below shows the receiving of a message.

Figure 3-8: Message receiving



3.6.4 Sending messages

After all processing of the segment sending UAP terminates (MHP termination or normal termination of SPP transaction), OpenTP1 sends, as messages, all the segments sent from the UAP at a time. Sending messages in this manner is called *asynchronous message send processing*. Use the function `dc_mcf_send()` [`CBLDCMCF('SEND ')`] for send-only messages. Use the function `dc_mcf_reply()` [`CBLDCMCF('REPLY ')`] for response messages.

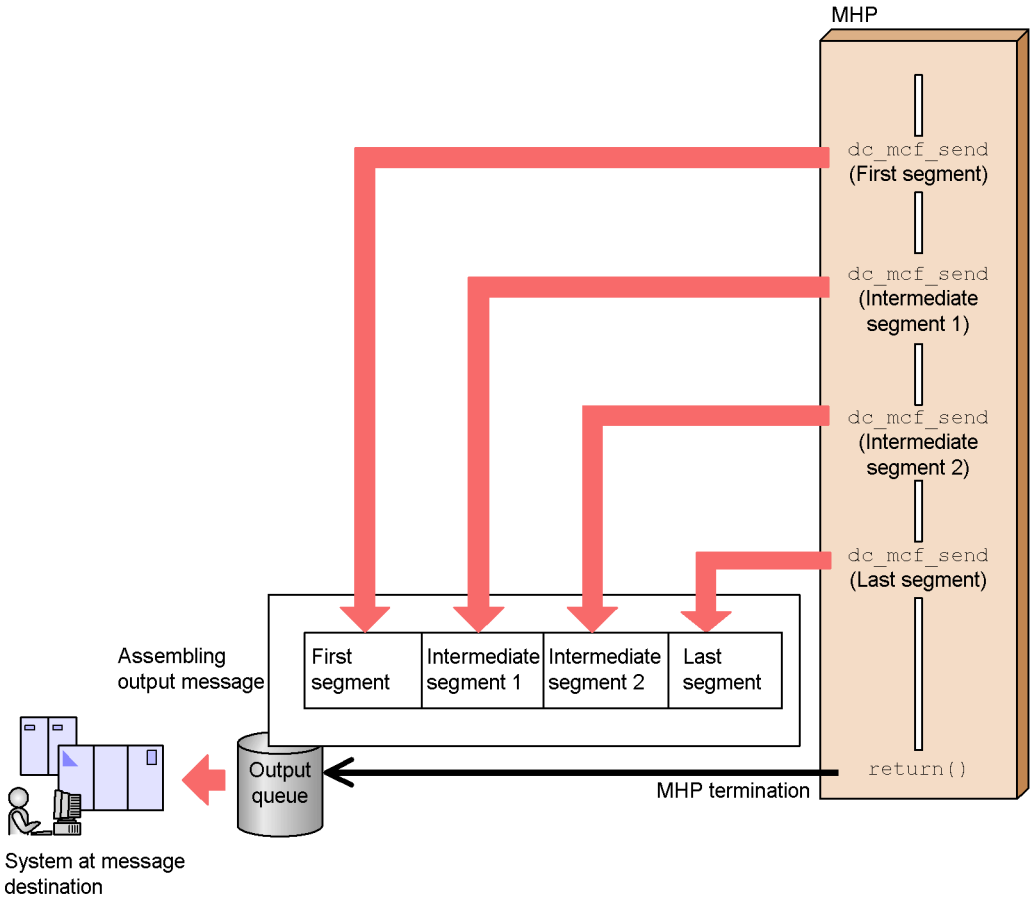
During asynchronous message send processing, if rollback processing is executed due to the following after the function has sent segments, all the segment send functions used from the UAP are invalidated:

- The UAP process terminates abnormally or message processing fails.

Before segments sent from the UAP are output, the user exit routine enables you to do processing such as editing of serial numbers or output messages.

The figure below shows message send processing.

Figure 3-9: Message send processing (asynchronous message sending)



3.6.5 Synchronous message processing

Use synchronous message processing to confirm the completion of the sending of messages during MHP processing or to synchronize UAP message exchange processing between systems. As for synchronous message exchange processing, send or receive processing is requested, the processing is completed, then the function called by the UAP returns.

(1) Types of synchronous message

The following functions are available for synchronous message exchange processing:

- Send function only for send processing
- Receive function only for receive processing

- Exchange function to execute send and receive processing consecutively

Synchronous message send processing

Use the function `dc_mcf_sendsync()` [`CBLDCMCF('SENDSYNC')`] to execute synchronous message send processing. When the UAP calls the function `dc_mcf_sendsync()`, the MCF writes a message to the output buffer (the output queue in memory), then sends the message to the own system. After the MCF confirms that the sending of the message to the own system is completed, the function `dc_mcf_sendsync` returns.

Synchronous message receive processing

When receiving a message from the own system, the MCF stores the message in the input buffer. The MHP calls the function `dc_mcf_recvsync()` [`CBLDCMCF('RECVSYNC')`] to receive the message.

If a message has been received from the own system, the message is passed to the function `dc_mcf_recvsync()`. If a message has not been received from the own system, the function `dc_mcf_recvsync()` continues waiting until a message is received. As soon as a message is received from the own system, the message is passed to the function `dc_mcf_recvsync()`.

Synchronous message exchange processing

Send processing and receive processing for synchronous messages can be done by one function. The MHP calls the function `dc_mcf_sendrecv()` [`CBLDCMCF('SENDRECV')`] to request the MCF to send a message. The MCF writes a message to the output queue, then sends the message to the own system. Even after send processing is completed, the function `dc_mcf_sendrecv()` does not return and proceeds to receive processing. The function `dc_mcf_sendrecv()` returns when receive processing is completed.

(2) Time monitoring of synchronous message processing

Monitoring time can be set to prevent the UAP from waiting for a response infinitely during synchronous message processing. Set the monitoring time for the argument `watchtime`. If 0 is specified, the synchronous exchange monitoring time specified in the UAP common definition of the MCF manager definitions is assumed to be specified. If 0 is defined as the monitoring time in the UAP common definition, the UAP waits for a response infinitely.

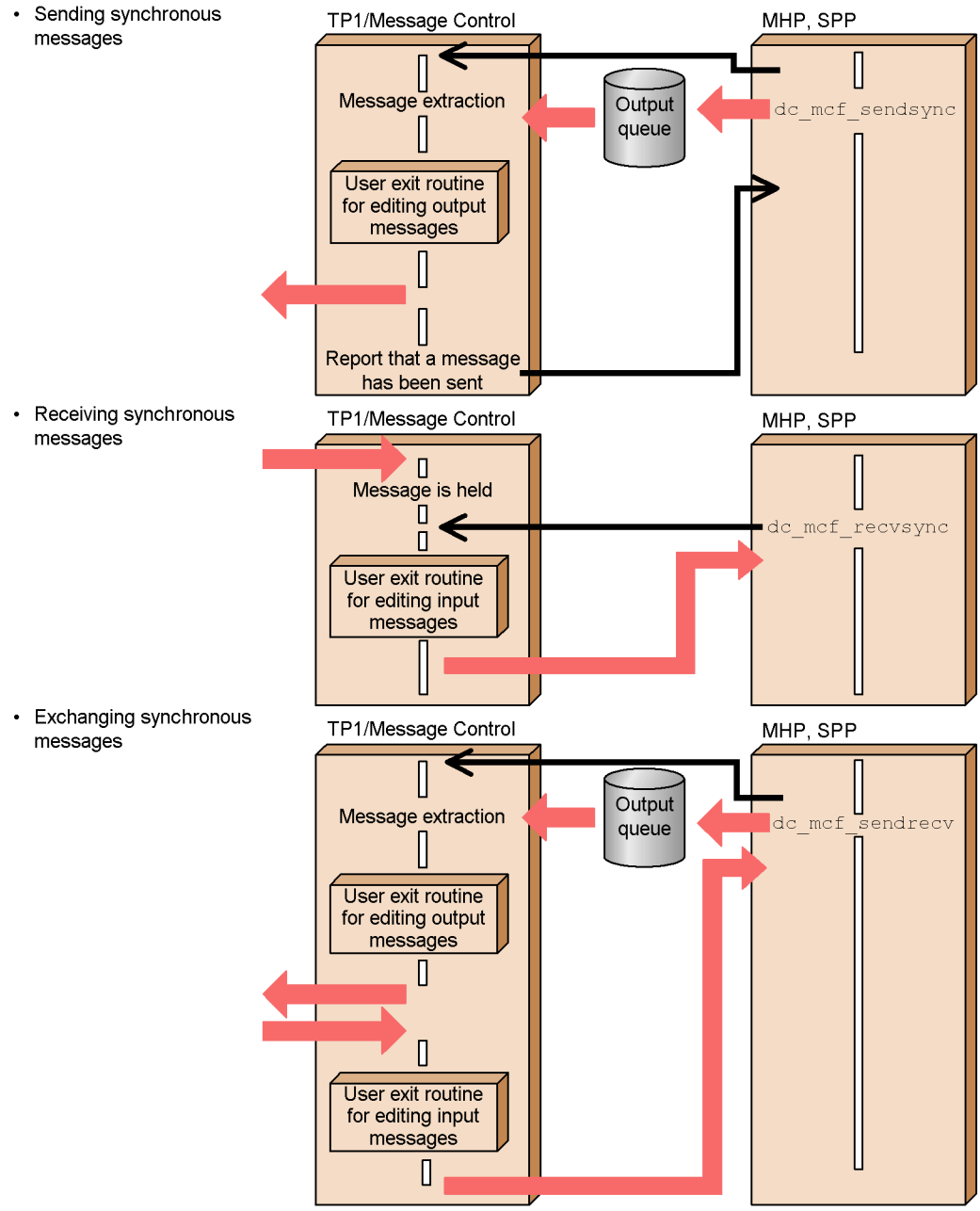
You can select whether to include the synchronous message processing time in the expiry time in a transaction branch. Specify this value using `trn_expiration_time_suspend` of the user service definition, user service default definition, and transaction service definition. You cannot include synchronous message processing time in a non-transactional MHP expiry time. For details on the value to be assigned to `trn_expiration_time_suspend` and transaction time monitoring, see the manual *OpenTP1 System Definition*.

(3) Synchronous message processing and rollback

If the MHP is rolled back, the synchronous message is not discarded. However, the message is discarded if multiple segments were sent by the function `dc_mcf_sendsync()` or `dc_mcf_sendrecv` and the `function return()` was called without the designation of the last segment (EMI).

The figure below shows synchronous message processing.

Figure 3-10: Synchronous message processing



3.6.6 Continuous-inquiry-response processing

Messages are transferred between a terminal and a UAP by continuing inquiry-response processing. Continuous inquiry-response processing can be executed by only MHPs for which continuous-inquiry-response type (`cont`) is specified as the application type.

(1) Outline of continuous-inquiry-response processing

An MHP executing continuous-inquiry-response processing calls the function `dc_mcf_receive()`, then receives a message from the terminal. After terminating processing, the MHP returns a response with the function `dc_mcf_reply()`. To switch to an MHP for continuous processing when a response is returned, specify the application name of the new MHP in the function `dc_mcf_reply()`. Without the application name specified, the previous MHP is started.

Also, the MHP, handling continuous-inquiry-response processing, can start an application by using the function `dc_mcf_execap()`. Only immediate start is permitted. In this case, only one MHP with cont specified can be started by executing the function `dc_mcf_execap()`. The MHP that started the application with `cont` specified cannot use the function `dc_mcf_reply()` because the continuous response right has moved from the MHP. Also, the MHP cannot use the function `dc_mcf_contend()`.

The function `dc_mcf_send()` (a send-only message to a terminal) can be used even during continuous-inquiry-response processing.

(2) Access to temporary-stored data

Temporary-stored data can be used during continuous inquiry-response processing. The temporary-stored data is used as information for transferring processing to the subsequent MHP to be started. Temporary-stored data can be used at a logical terminal. Thus, continuous-inquiry-response processing can be carried out by using one MHP shared by multiple logical terminals.

Allocate an update area and a recovery area as temporary-stored data areas in the shared memory. For each MHP, specify the length of the temporary-stored data storage area in the MCF application definition.

Temporary-stored data can be used only when the continuous-inquiry-response mode is enabled. Temporary-stored data cannot be used in other message communication modes.

(a) Receiving temporary-stored data

Call the function `dc_mcf_tempget()` [`CBLDCMCF('TEMPGET')`] to use temporary-stored data from an MHP. The function `dc_mcf_tempget()` is executed on the assumption that there is $(00)_{16}$ of the length specified in tempsize of the MCF application attribute definition in the following cases:

- The temporary-stored data storage area is in initial state.
- There is no temporary-stored data.

If the receive area length specified in the function `dc_mcf_tempget()` is shorter than the length of temporary-stored data, only the portion of temporary-stored data equivalent to the specified length is received. The excess portion is truncated. If the receive area length specified in the function `dc_mcf_tempget()` is larger than the temporary-stored data length, only the temporary-stored data is stored in the receive area.

(b) Updating temporary-stored data

To update temporary-stored data, use the function `dc_mcf_tempput()` [`CBLDCMCF('TEMPPUT')`]. When the temporary-stored data, area for storing is updated, the data itself is replaced. A value exceeding the value specified in the MCF application definition cannot be set as the length of update area.

Call the function `dc_mcf_tempget()` before the function `dc_mcf_tempput()`. Otherwise, the `dc_mcf_tempput()` returns with an error.

(3) Terminating continuous-inquiry-response processing

Continuous-inquiry-response processing terminates when one of the events shown below is executed. The temporary-stored data storage area which has been used is deleted when continuous-inquiry-response processing terminates.

- The function `dc_mcf_contend()` [`CBLDCMCF('CONTEND ')`] is called from the MHP.
- The logical terminal name is specified in the `mcftendct` command in order to forcibly terminate continuous-inquiry-response processing.

  The `mcftendct` command can also be called by using the function `dc_adm_call_command()` from a UAP which is not involved in continuous-inquiry-response processing.
- The UAP terminates abnormally.

  The UAP terminates abnormally if the MHP that executed continuous-inquiry-response processing does not use the function `dc_mcf_reply()`.

(4) Processing if an error event occurs due to UAP abnormal termination

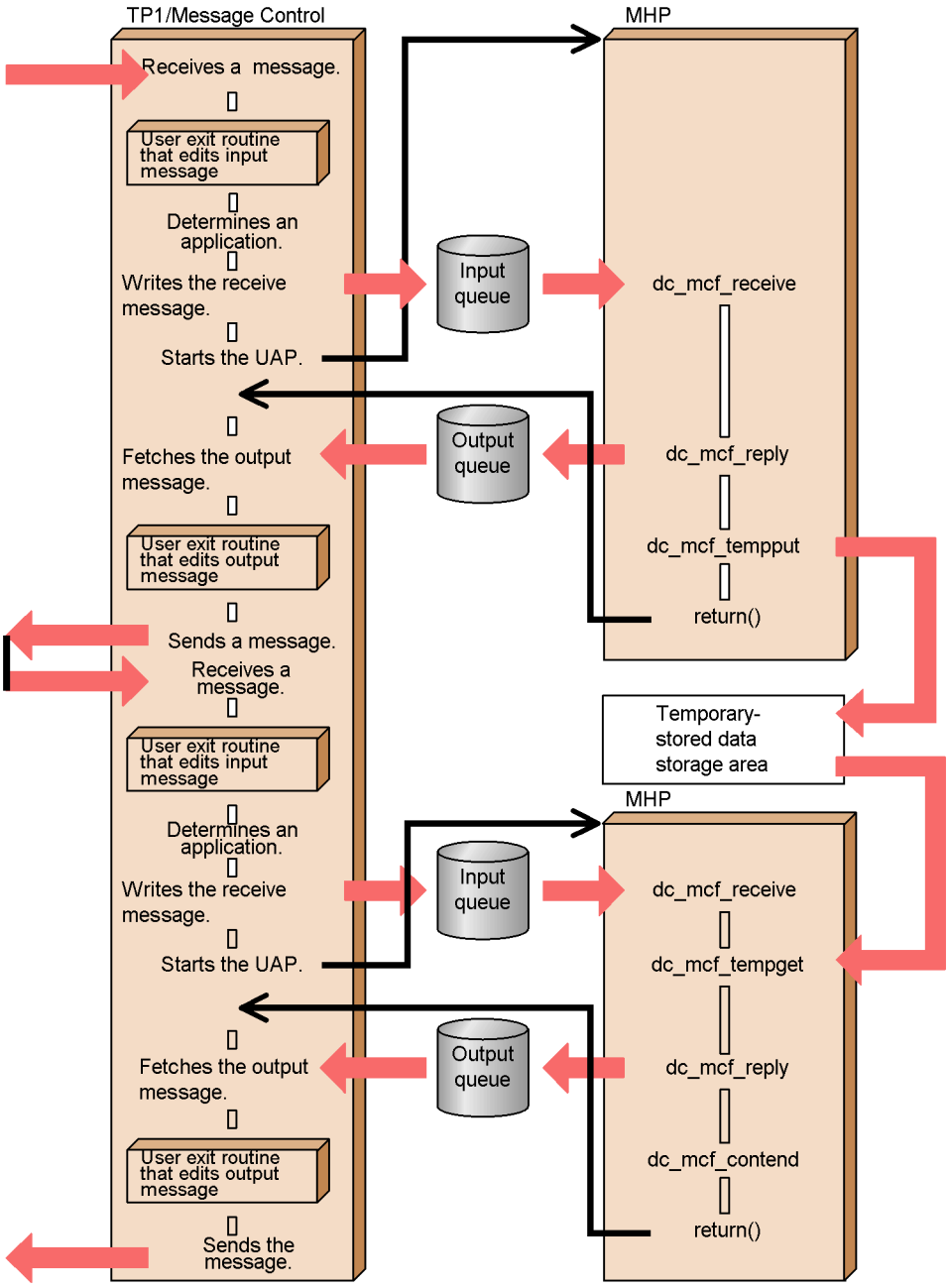
If the UAP terminates abnormally during continuous-inquiry-response processing, `ERREVT3` is reported. The continuous-inquiry-response processing can be continued by the following:

- Use the MCF event handling MHP corresponding to this `ERREVT3` in order to use the function `dc_mcf_reply()` in which the name of the next application to be started is specified. The continuous-inquiry-response processing terminates if the

function `dc_mcf_reply()` is not called.

The figure below shows continuous-inquiry-response processing.

Figure 3-11: Outline of continuous-inquiry-response processing



3.6.7 Resending messages

Sent messages can be resent using the function `dc_mcf_resend()` [`CBLDCMCF('RESEND  ')`]. A resent message is treated as a new message separate from the message that was sent in the past. Messages are resent in the following cases:

- When a sent message is printed, characters are unclear and damaged printed.
- Multiple copies of a document are necessary.
- The screen showing a message is cleared.

(1) Conditions for resending messages

The following messages can be resent:

- Sent messages that have been assigned output sequence numbers
- Sent messages that remain in the output queue
- Messages sent to terminals

If the message to be resent is not found in the message queue (disk queue), the function `dc_mcf_resend()` returns with an error.

(2) Specification for messages to be resent

Messages to be resent are identified using the following information which was specified on the sent message:

- Output destination logical terminal name

  The output destination logical terminal name determines the output queue which contains the message to be selected.

- Message output sequence number

  Output sequence numbers can be set in one of the following ways. When resending messages, it is possible to give them new output sequence numbers:

  1. Output sequence number of the message to be resent
  2. Specification that, of all the sent messages, the message with the last output sequence number is to be resent.

- Message type (general branch or priority branch)

  When a message is resent, its message type can be newly specified.

(3) Relationship with network communication definition

Resending of a message requires the use of a work area with the size equal to the maximum segment length specified for the `-e` option to the UAP common definition (`mcfmuap`) included in the MCF manager definition. If the segment of the message being resent is larger than this work area, the function `dc_mcf_resend()` returns

with an error. Therefore, the value specified for the -e option to the UAP common definition must be at least the maximum length of the message to be resent.

More than one message with the same sequence number may be present in the message queue file, depending on the sequence number specification given in the -l option to the UAP common definition included in the MCF manager definition. In this case, which message is resent is unpredictable.

3.7 MCF transaction control

OpenTP1 can treat processing of a message from the remote system as a transaction.

This section explains transaction control of message exchange application programs (MHPs). For details on transaction control of client/server mode UAPs (SUPs and SPPs), see *2.3 Transaction control*.

3.7.1 MHP transaction control

An MHP always behaves as a transaction during a period from the MHP start (when OpenTP1 receives a message) to the MHP termination. This means that OpenTP1 treats all MHP processing as transactions.#

MHP service functions cannot use a transaction control function (`dc_trn_begin()` or other synchronization point acquisition function beginning with `dc_trn`). Also, if a service request is called from an MHP to an SPP, the SPP cannot use the transaction control function. When requesting a service from an MHP to an SPP, verify that the transaction control function has not been called in the SPP.

#

When an MCF extended facility is used, MHP processing is not treated as a transaction. Such an MHP is called a *nontransaction attribute MHP*. For details on nontransaction attribute MHP, see *3.8.3 Nontransaction attribute MHP*.

(1) Specification of transaction attribute

For MHPs, the user service definition must include the specification of `atomic_update=Y` indicating that the MHP has the transaction attribute.

(2) MHP's synchronization point acquisition

During MHP processing, the synchronization point can be acquired as a commitment in chained mode. To acquire the synchronization point, call the function `dc_mcf_commit()` [`CBLDCMCF('COMMIT  ')`]. When the function `dc_mcf_commit()` returns, the subsequent MHP process becomes a global transaction.

Suppose that a global transaction beginning with an MHP consists of more than one transaction branch (the MHP calls the SPP with the function `dc_rpc_call()`). Unless the processing result of each transaction branch brings about a commitment, no commitment comes into effect. If the global transaction is not committed, all transaction branches are rolled back.

Before a message is received, the synchronization point cannot be acquired using the function `dc_mcf_commit()`. In addition, once the synchronization point is acquired using the function `dc_mcf_commit()`, the MHP can no longer receive the message.

Message processing for which synchronization point acquisition using the function dc_mcf_commit() is performed is the activation of an asynchronous message and an application. Processing for sending or receiving a synchronous message is not the target of synchronization point acquisition.

The function dc_mcf_commit() can be used only from MHPs for which the nonresponse type (noans type) is specified in the MCF application definition. If the function is called from an MHP of another type, it returns with an error. UAPs other than MHPs cannot call the function dc_mcf_commit().

(3) MHP rollback processing

(a) If MHP processing terminates abnormally:

If an MHP terminates abnormally or rolls back,#1 an error event is generated. The type of the error event depends on whether the function dc_mcf_receive() has received the first segment.

- ERREVT2#2: The MHP terminated abnormally before the function dc_mcf_receive() received the first segment.
- ERREVT3: The MHP terminated abnormally after the function dc_mcf_receive() received the first segment.

#1

Excludes cases in which r is specified for the recvmsg operand in the MCF application definition (mcfaalcap -g) or in which DCMCFRTRY or DCMCFRRTN is specified for action of the function dc_mcf_rollback().

#2

When a non-resident MHP cannot start for a reason such as those given below, ERREVT2 is not reported.

- The corresponding load module does not exist.
- There is no service function corresponding to the entry point defined in the RPC interface definition file.

In this case, the system shuts down the schedule of the input queue for the relevant service group and leaves a receive message in the input queue.

(b) If an error occurs during MHP processing:

If MHP transaction processing ends up with an error, call the function dc_mcf_rollback() [CBLDCMCF('ROLLBACK')] from the MHP in order to return to the status before the message was received. If the MHP that received the message was rolled back, OpenTP1 decides whether to reschedule the MHP according to the argument specification in the function dc_mcf_rollback().

- If NORETURN (DCMCFNRTN for action) is specified:

After the rollback, control does not return to the MHP. The MHP terminates abnormally and ERREVT3 is reported.

- If RETURN (DCMCFRRTN for action) is specified:

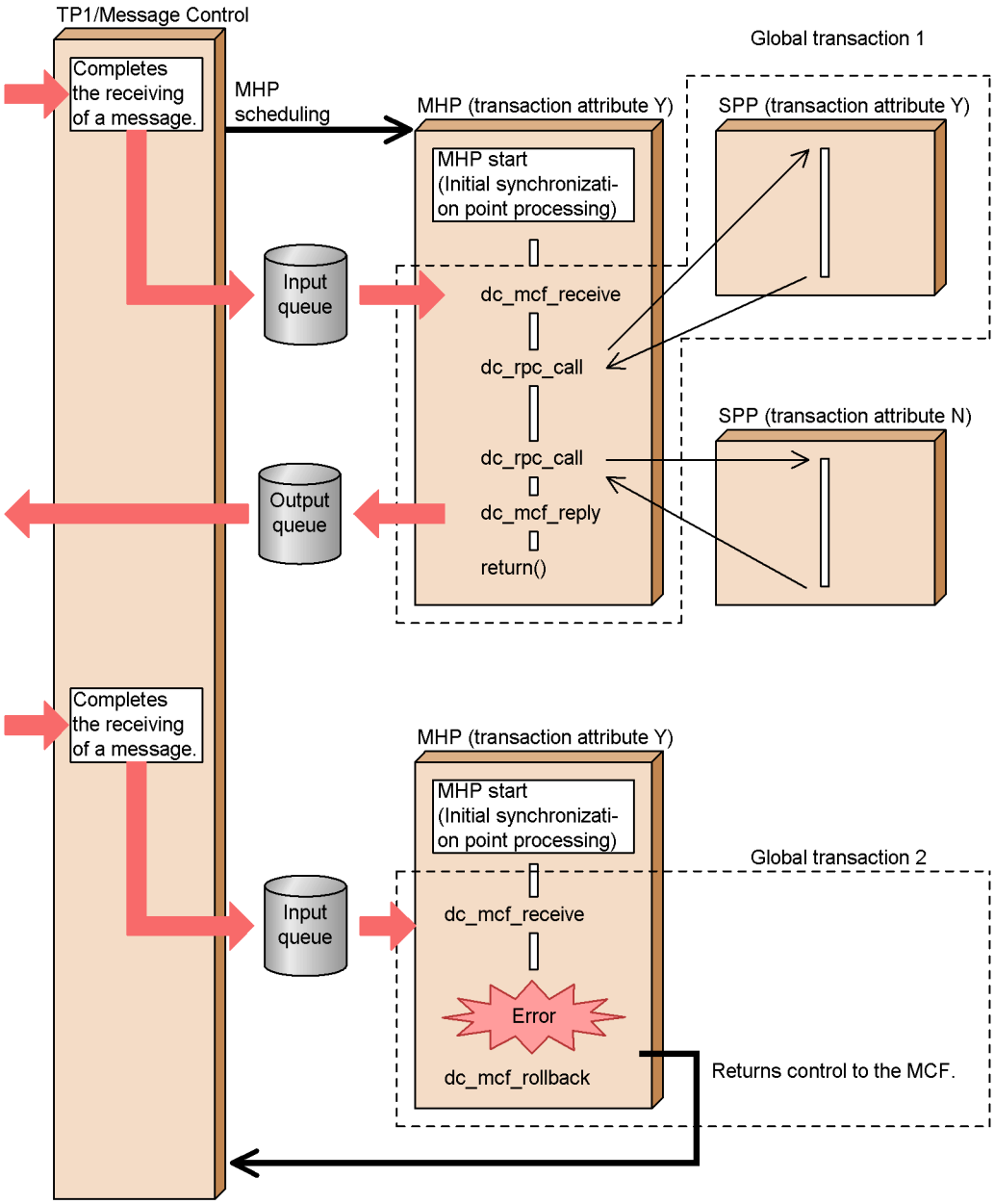
  If the rollback is successful, the function dc_mcf_rollback() returns. Thereafter, the MHP can continue any processing. After the rollback, a new separate transaction comes into effect.

- If RETRY (DCMCFRTRY for action) is specified:

  When the function dc_mcf_rollback() does not return, the MHP terminates the process. After the rollback, the MHP is rescheduled. Before the function dc_mcf_rollback() can be used for setting this value, an application startup process must be present on the node.

The figure below shows the relationship between message exchange processing and transactions.

Figure 3-12: Relationship between message exchange processing and transactions



Explanation:

1. When TP1/Message Control receives a message, processing started by the MHP becomes a global transaction.
2. If an error occurs during MHP transaction processing, control is returned to the MCF after rollback processing (partial recovery) is executed.

   New transaction processing can be executed by using the function `dc_mcf_rollback()` in which return is specified (`DCMCFRRTN` set in `action`).

(4) If an MHP is committed when the message exchange function returns with an error:

If MHP processing is terminated because the message exchange function returns with an error, the transaction itself might be committed. If the resource manager (RM) has been accessed (DAM, TAM) in the MHP processing, this access processing is committed. To roll back the access processing, call the `abort` function or create processing for using the function `dc_mcf_rollback()` after an error is returned.

(5) Using the transaction start function from an MHP

Even MHPs can use a transaction starting function (`dc_trn_begin()`) if the function is outside the service function processing range (within the main function processing range). The transaction start function and commitment functions can be called between main functions (e.g., between the `dc_rpc_open()` and the `dc_mcf_mainloop()` or between the `dc_mcf_mainloop()` and the `dc_rpc_close()`).

If the function `dc_trn_begin()` is called as an MHP main function, acquire a synchronization point by using the function `dc_trn_unchained_commit()` (commitment in unchained mode) as an MHP main function.

3.8 MCF extended facilities

The following MCF facilities are also supported in addition to the message exchange facility:

- Starting application programs
- MHP startup using command
- MHP with nontransaction attribute
- Time monitoring with the facility for user timer monitoring

3.8.1 Starting application programs

An MHP or SPP can be started from another MHP or an SPP. In the function `dc_mcf_execap()` [`CBLDCMCF('EXECAP  ')`] (for starting application programs), specify the application name of the MHP or SPP to be started and the message segment to be transferred.

(1) MCF processes used for starting application programs

When the application active facility (`dc_mcf_execap()`) is in use, an MCF process separate from the message exchange functions (such as `dc_mcf_receive()` and `dc_mcf_send()`) is used. The MCF process used for message exchanging is called an MCF communication process, whereas an MCF process used by the function `dc_mcf_execap()` is called an *application startup process*. Application startup processes do not depend on the communication protocol.

(2) How to start application programs

Only MHPs and SPPs can be started with the function `dc_mcf_execap()`. MHPs can be started by calling the function `dc_mcf_execap()`.

(a) Ordinary starting of application programs (starting MHPs)

Segments sent from the function `dc_mcf_execap()` can be received with the function `dc_mcf_receive()` called by an MHP. MHPs can be started only if they exist in the same node as the UAP that called the function `dc_mcf_execap()`. MHPs at other nodes cannot be started using the function `dc_mcf_execap()`.#

#

There is no restriction when communicating by using TP1/NET/HNA-NIF, because message exchanging is done with the function `dc_mcf_execap()`.

(3) Time to start

The MHP or SPP which has been designated for activation will actually start when:

- If the MHP has called the function `dc_mcf_execap()`:

The MHP transaction is committed (the MHP normally returns or the function `dc_mcf_commit()` normally returns).

- If the SPP has called the function `dc_mcf_execap()`:

  The transaction is committed.

  If the SPP has called the function `dc_mcf_execap()`, the prerequisite condition is that the SPP is working as a transaction and that the main function of the SPP calls the function `dc_mcf_open()`.

(4) How to start application programs

An MHP or SPP can be started by either of the following methods:

(a) Immediate start

The application program is started immediately after the UAP process which called the function `dc_mcf_execap()` is committed.

(b) Timer start

The application program is started at the specified time after the function `dc_mcf_execap()` is called. Timer starts in either of the following two ways:

- Interval timer start

  The application program is started a specified number of seconds after the function `dc_mcf_execap()` is called. If the UAP process which called the function `dc_mcf_execap()` is not committed after the specified number of seconds, the application program will be started when commitment occurs.

- Time point timer start

  The application program will be started when the specified time comes after the function `dc_mcf_execap()` is called.

  If the time the function `dc_mcf_execap()` called is later than the time specified in the function, the application program is immediately started or will be started at the specified time on the following day, depending on the specification given in the UAP common definition included in the MCF manager definition.

  When the application active facility is used, note the following: if the time schedule is changed from Standard to Daylight Saving or vice versa during the period from activation request issuance to the scheduled UAP start time, the UAP will be activated based on the schedule that was in use when the activation request was issued.

(5) Error event if an error occurs before an application program is started

After the function `dc_mcf_execap()`, if an error occurs before the MHP or SPP is started, the following MCF events are generated:

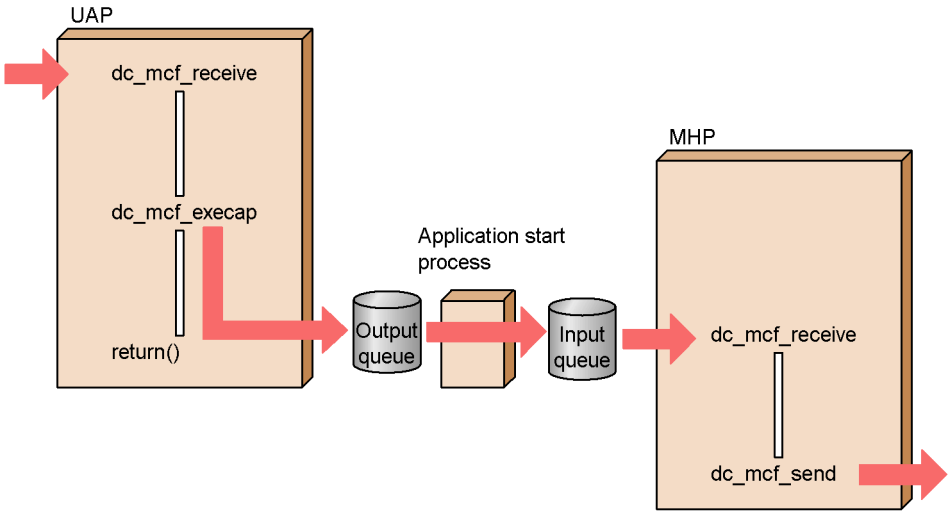
- With immediate start: ERREVT2
- With timer start: ERREVT4

For details on the error events, see *3.10 MCF events*.

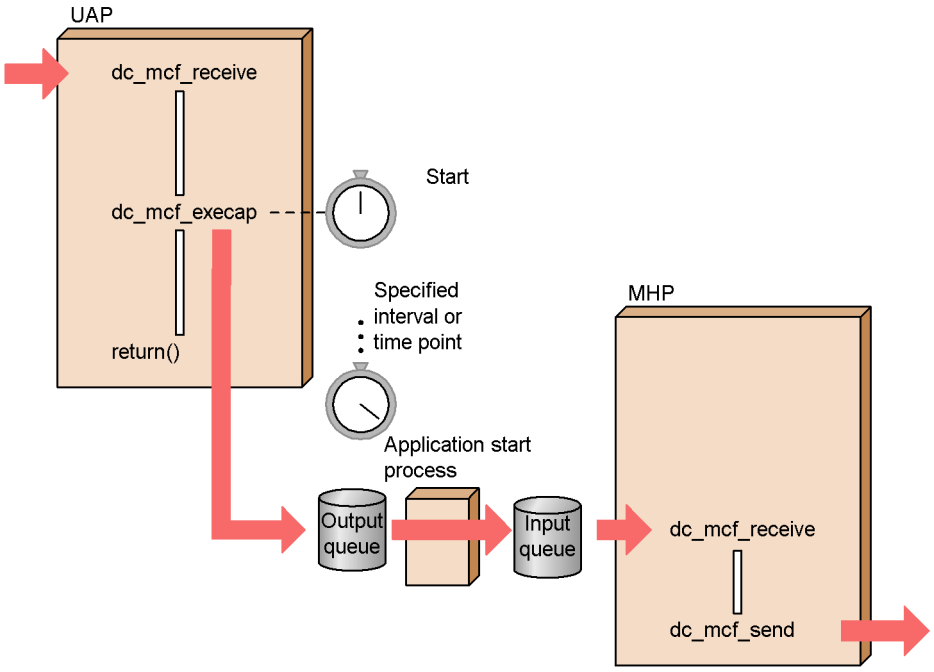
The figure below shows how to start an application program.

Figure 3-13: How to start application program

● Immediate start (start after the UAP returns normally)



● Timer start (MHP is started a specified time after the calling of the function dc_mcf_execap())



(6) Network communication definition

(a) MCF communication configuration definition

In addition to an ordinary execution process, an application startup process is required for the node at which the UAP that calls the function `dc_mcf_execap()` exists. Specify the application startup process in the application start environment definition. With OpenTP1 which uses the application program start function, create the application startup environment definition of the MCF communication configuration definitions.

(b) MCF application definition

The application type specified in the type operand of the application attribute definition (`mcfaalcap`) in the MCF application definitions determines which type of MHP is to be started.

- When starting a response type (`ans`) MHP:

  Only an MHP with the response type (`ans`) specified can send response messages. When an `ans` type MHP is started from the MHP that received an inquiry message, the response right is transferred. Because of this, the `ans` type MHP can be started only once. Response messages cannot be sent from the MHP that started the `ans` type MHP. The `ans` type MHP cannot also be started from an MHP which has sent a response message.

  An `ans` type MHP cannot be started from an SPP.

- When starting a nonresponse type MHP (`noans` specified):

  An MHP with the nonresponse type (`noans`) specified can be started more than once from one transaction.

- When starting a continuous-inquiry-response type MHP (`cont` specified):

  An MHP with the continuous-inquiry-response type (`cont`) specified can be started only from an MHP handling continuous-inquiry-response processing. In this case, only immediate start is permitted (timer start is not permitted). Only one `cont` type MHP can be started if the function `dc_mcf_execap()` is called from an MHP handling continuous-inquiry-response processing. The MHP that started a `cont` type application cannot call the function `dc_mcf_reply()` because the continuous response right has been transferred from the MHP. Also, the MHP cannot call the function `dc_mcf_contend()`.

(7) Input source logical terminal name to be passed to the MHP to be started

When an MHP starts another MHP by using the function `dc_mcf_execap()`, the started MHP receives the name in the first-received message as the logical terminal name of the message input source. Also, when the function `dc_mcf_execap()` is called from the MHP, the name in the first-message is passed as the logical terminal name of the message input source.

When an SPP starts an MHP by using the function `dc_mcf_execap()`, the started MHP receives * as the logical terminal name of the message input source. Also, when the function `dc_mcf_execap()` is called from the MHP, * is received as the logical terminal name of the message input source.

Figures 3-14 to 3-17 show how application programs are started, and specification of the `type` operand.

Figure 3-14: Starting MHP from MHP that received send-only message

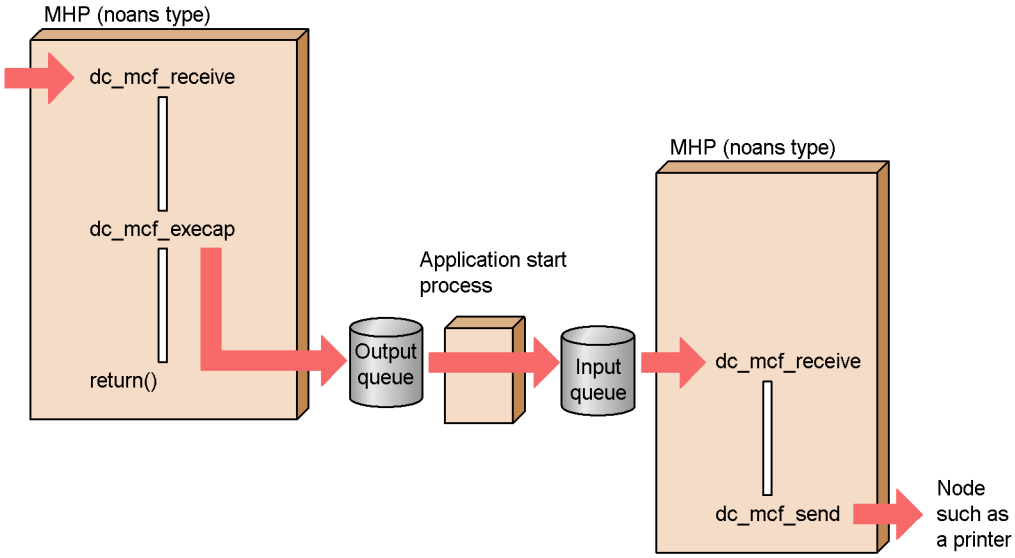
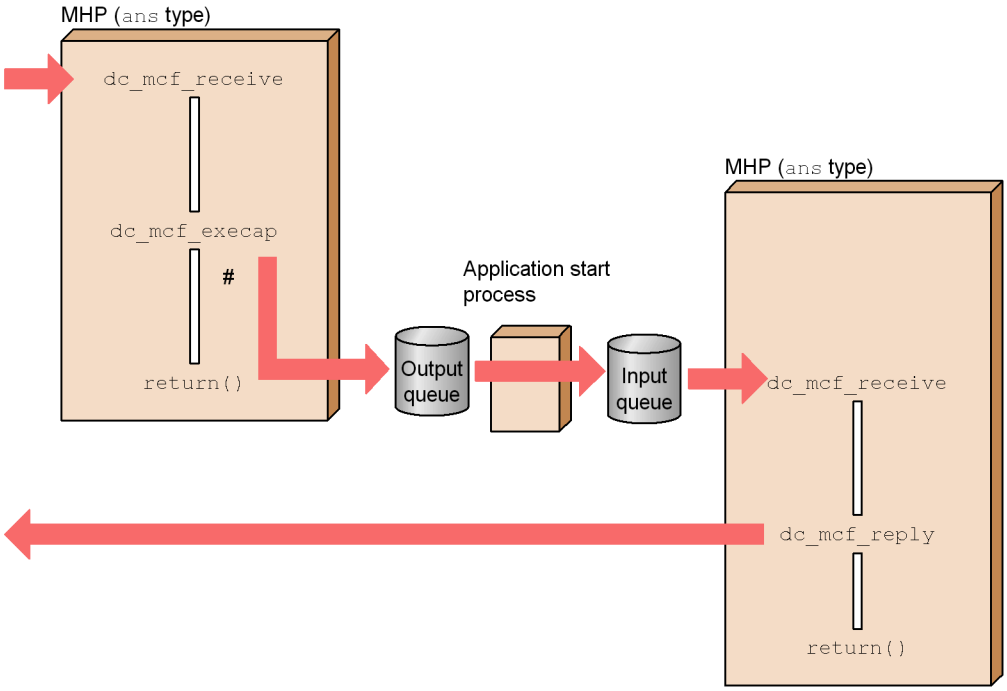


Figure 3-15: Starting MHP from MHP that received inquiry-response message



#: If the function dc_mcf_reply is called after the start of an ans-type MHP is requested, an error is returned.

Figure 3-16: Starting MHP, which sends send-only message, from MHP that handles inquiry-response message processing

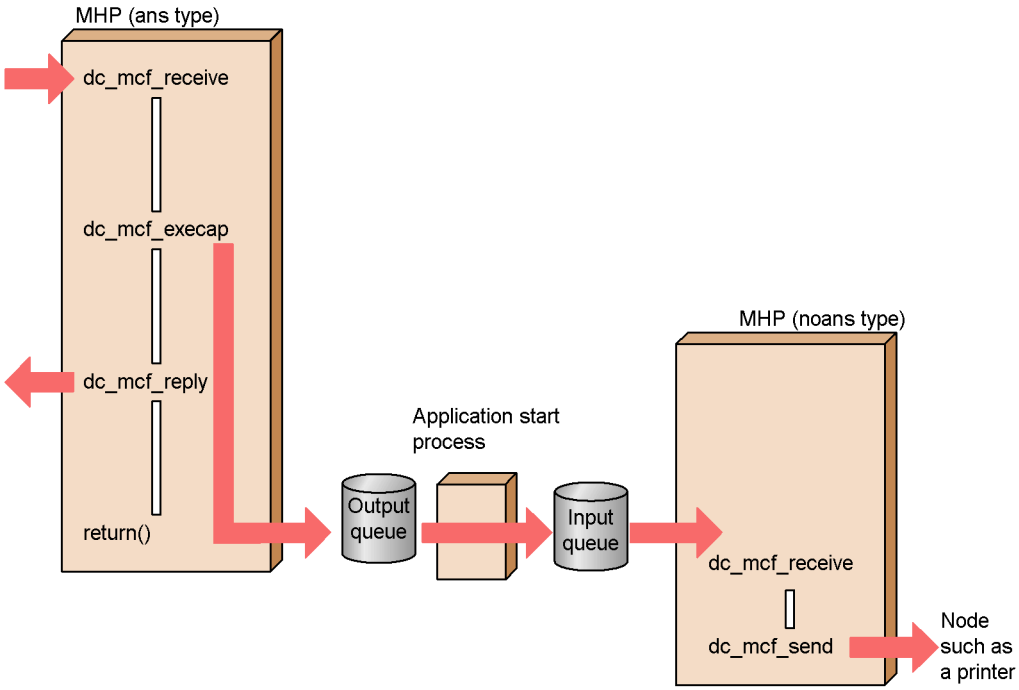
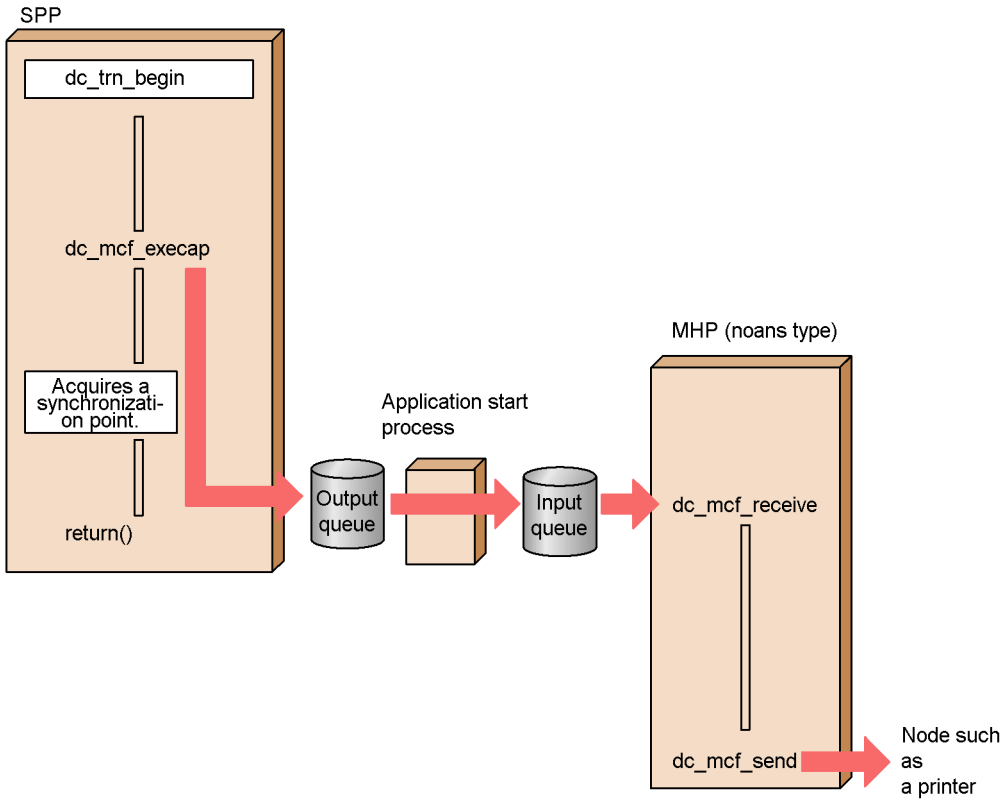


Figure 3-17: Starting MHP from SPP handling transaction processing



(8) Handling of timer start upon TP1/Message Control rerun

Explained below is the handling of timer start upon an OpenTP1 rerun subsequent to a fault which occurs during wait for timer start. After an OpenTP1 rerun, timer start can be inherited only when the disk queue is in use. If a rerun occurs, the function `dc_mcf_execap()` to be timer-started is handled as follows:

(a) Definition of timer start inheritance

If `reruntm=yes` is specified for the `-o` option to the mcftpsvr definition command for the MCF communication configuration definition, the timer start message before the rerun is inherited. If the time specified in the function `dc_mcf_execap()` has already come, the timer start message is inherited as an immediate start message. Otherwise, the application program will be started at the specified time.

If `reruntm=no` is specified, timer start is not inherited once a rerun occurs. The timer-started function `dc_mcf_execap()` must be called from the UAP.

(b) User exit routines for modifying conditions for timer start inheritance

Definition of timer start inheritance user exit routines can be used to modify conditions for timer start inheritance. These user exit routines are called *exit routines for determining timer start inheritance*. Before a user exit routine for determining timer start inheritance can be used, `reruntm=yes` must be specified for the `-o` option to the `mcftpsvr` definition command for the MCF communication configuration definition.

For details on user exit routines for determining timer start inheritance, see *3.9.2 User exit routine that determines the inheriting timer-start message*.

3.8.2 MHP startup using command

MHPs can be activated using OpenTP1 command (`mcfuevt` command). Even if an MHP is usually started by message reception, it can be activated by `mcfuevt` command to become ready for sending messages to other systems.

Only nonresponse type (`noans` type) MHPs can be activated by `mcfuevt` commands. If an MHP is to be activated by `mcfuevt` command, specify the `noans` type for the MHP.

(1) Definition of MHPs to be activated by command

Suppose that the application name of the MHP to be activated by `mcfuevt` command is `UCMDEVT`. Specify the following values for the `-n` option to the `mcfaalcap` operand for the application attribute definition in the MCF application definition:

`name` operand: `UCMDEVT`

`kind` operand: `user` (optional)

`type` operand: `noans` (optional)

(2) How to start MHP

The `mcfuevt` command is executed to start an MHP. The MCF communication process identifier and the input message to be passed to the MHP are specified as the arguments to the `mcfuevt` command.

If the `mcfuevt` command is executed before `UCMDEVT` is defined, it returns with an error. In this event, `ERREVT1` does not inform.

Since MHPs activated by commands do not depend on the communication protocol, it is recommendable that an application startup process should be specified for the MCF communication process specified in the `mcfuevt` command.

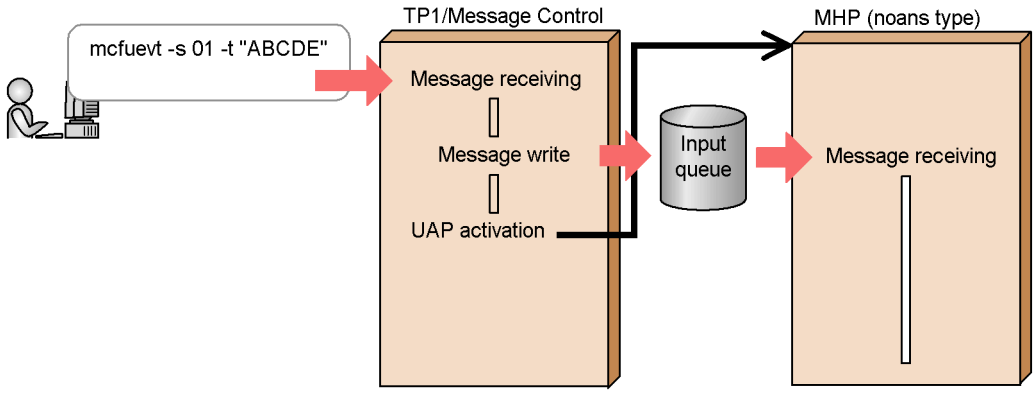
(3) Input source logical terminal name and connection name of MHP activated by command

The input source logical terminal name of an MHP activated by `mcfuevt` command is `@UCEV`*xxx*, where *xxx* is the MCF process identifier. If a message is sent from a UAP to this input source logical terminal name, the function returns with an error.

The connection name is ********.

The figure below shows how an MHP is activated by command.

Figure 3-18: MHP activation by operation command



3.8.3 Nontransaction attribute MHP

MHPs which do not work as a transaction (the nontransaction attribute MHPs) can be created. The nontransaction attribute MHPs cannot be treated as transactions, but assure a higher processing speed than ordinary MHPs.

(1) Difference from MHPs working as transactions

The nontransaction attribute MHPs can use message exchange functions like MHPs which work as transactions, but the following differences are involved:

- A message output sequence number can be specified, but it is not eligible for error recovery.
- The nontransaction attribute MHPs cannot use synchronization point processing functions (`dc_mcf_commit()` and `dc_mcf_rollback()`) and cannot call the message resend function (`dc_mcf_resend()`). If one of these functions is called, it returns with an error.

(2) Definition of nontransaction attribute MHPs

(a) Available message queue

The nontransaction attribute MHPs can use memory queues, but cannot use disk queues. Specify the memory queue in the `quekind` operand that is the `-g` option to the `mcfaalcap` operand for the application attribute definition in the MCF application definition.

(b) Transaction attribute of MHPs

For the nontransaction attribute MHPs, specify `nontrn` in the `trnmode` operand for

the application attribute definition in the MCF application definition. The MHP is not treated as a transaction even if `atomic_update=Y` is specified in the user service definition.

(c) Time monitoring

Time monitoring of the nontransaction attribute MHPs is specified by the `-v` option to the `mcfaalcap` operand for the application attribute definition in the MCF application definition. When the specified time comes, the nontransaction attribute MHP terminates abnormally. If `0` is specified in this definition, time monitoring is not in effect.

If a synchronous message exchange request is used from a nontransaction attribute MHP, the processing time for this message exchange is not included in the time monitored. If a nontransaction attribute MHP requests a service to an SPP, the SPP processing time is also included in the time monitored (if the SPP processes a synchronous message, the processing time is included in the time monitored).

(3) If an error occurs in the nontransaction attribute MHP:

The MHP for handling MCF event `ERREVT2` or `ERREVT3` is activated depending on the specification in the MCF application definition. If the nontransaction attribute MHP is requesting an SPP for service, nothing is done on the SPP process.

If temporary-stored data cannot be actually updated during processing in continuous inquiry response mode, the processing in continuous inquiry response mode is terminated regardless of the error event definition. If the processing in continuous inquiry response mode cannot be terminated because of an internal error or other condition, a message log is output to prompt the execution of the command for force termination of continuous inquiry response (`mcftendct -f`). Execute this command to terminate processing in continuous inquiry response mode.

3.8.4 Time monitoring with the facility for user timer monitoring

You can use a function from an MHP or SPP to set time monitoring and to cancel the setting. This facility is called the *facility for user timer monitoring*. It enables you to monitor a desired time. To use the facility for user timer monitoring, you must specify `usertime=yes` in the `-p` option of the MCF communication configuration definition `mcfttim`.

To set user timer monitoring, call the function `dc_mcf_timer_set()` [`CBLDCMCF('TIMERSET')`]. To cancel a user timer monitoring, call the function `dc_mcf_timer_cancel()` [`CBLDCMCF('TIMERCAN')`]. Processing for setting and canceling user timer monitoring is run when the function is called regardless of transactions.

At a fixed time monitoring interval, the MCF checks whether timeout has occurred. Specify the time monitoring interval in the `btim` operand of the `-t` option of the MCF communication configuration definition `mcfttim`.

If timeout has occurred, OpenTP1 starts the MHP specified in the arguments of the function `dc_mcf_timer_set()`. Specifying user data in the arguments of the function `dc_mcf_timer_set()` causes OpenTP1 to pass the data as a message to the MHP started after timeout occurs.

The `mcftlsutm` command can be used to display the user timer monitoring status. For details about the `mcftlsutm` command, see the manual *OpenTP1 Operation*.

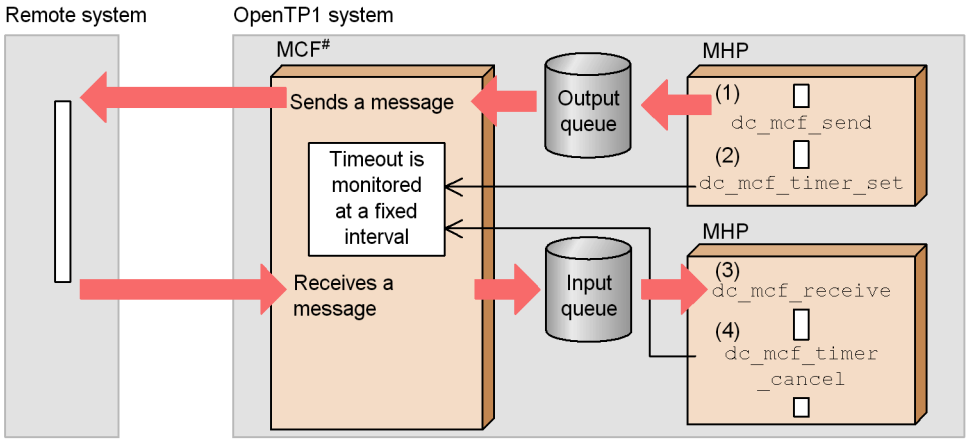
The facility for user timer monitoring can be used under any protocol.

(1) Example

The figure below gives an example of using the facility for user timer monitoring. This example shows how the time of responses from a remote system is monitored.

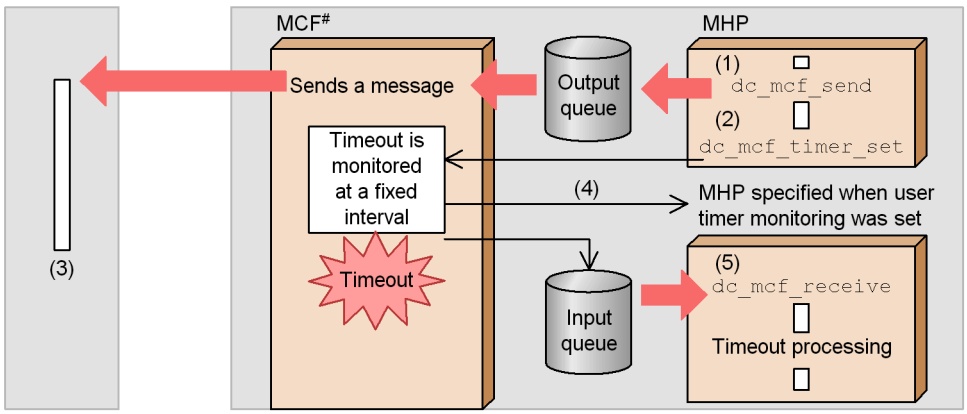
Figure 3-19: Example of using the facility for user timer monitoring

• When a response is received within the user timer monitoring time



(1) Sends a message to the remote system.
(2) Sets up user timer monitoring by calling the function `dc_mcf_timer_set`.
(3) Receives a response message from the remote system.
(4) The applicable user timer monitoring is cancelled with the function `dc_mcf_timer_cancel`.

• When no response is received within the user timer monitoring time



(1) Sends a message to the remote system.
(2) Sets up user timer monitoring by calling the function `dc_mcf_timer_set`.
(3) No response message has been sent from the remote system for some reason.
(4) Since the MCF detected a timeout, it starts the MHP that was specified when user timer monitoring was set.
(5) If user data is specified for the argument of the function `dc_mcf_timer_set`, that data is transferred to the MHP as a message.

#: MCF: Generic term for TP1/Message Control, TP1/NET/Library, and communication protocol products

(2) Notes on using the facility for user timer monitoring

1. User timer monitoring is set or canceled when the relevant function is called. Therefore, processing for setting or canceling user timer monitoring is not disabled even if the transaction is rolled back.
2. The MHP to be started upon occurrence of timeout must be a nonresponse-type (`noans` type) MHP. If the MHP specified in the arguments is not nonresponse-type, the function `dc_mcf_timer_set()` called from an MHP or SPP for setting a user timer monitoring returns an error.
3. Since OpenTP1 monitors timeout at fixed intervals, an error occurs between the monitoring time specified when the user timer monitoring was set and the time that elapses before actual detection of timeout.
4. If the function `dc_mcf_timer_cancel()` is called immediately before the MHP is started due to timeout, the function may return an error with the message *Timeout occurred* and the MHP may start.
5. If timeouts occur frequently while you are using the user timer monitoring facility, the performance of normal message control processing is affected. Do not set up normal processing so that an application starts when timeout occurs.
6. You must specify the maximum number of requests allowed for running a user timer monitoring in the `timereqno` operand in the `-p` option of the communication configuration definition `mcfttim`. Before processing starts, the MCF allocates the same number of monitoring tables as the number of requests specified in this operand. The tables are allocated on static shared memory. Setting one value requires static shared memory equivalent to *about 100 bytes + user data size*. Add the total capacity of static shared memory on all MCFs to the `-p` option of the MCF manager definition `mcfmcomn` and the `static_shmpool_size` operand of the system environment definition.
7. If the system goes down while time monitoring is in progress, monitoring is disabled when the system restarts (at rerun). However, if a disk queue is being used as the input queue and the system goes down immediately before the MHP is started due to timeout, the MHP may start after the system restarts. Therefore, we recommend using a memory queue as the input queue.
8. Note above also applies when the MCF is restarted (rerun) individually.
9. You cannot set or cancel user timer monitoring for MCFs on other nodes.

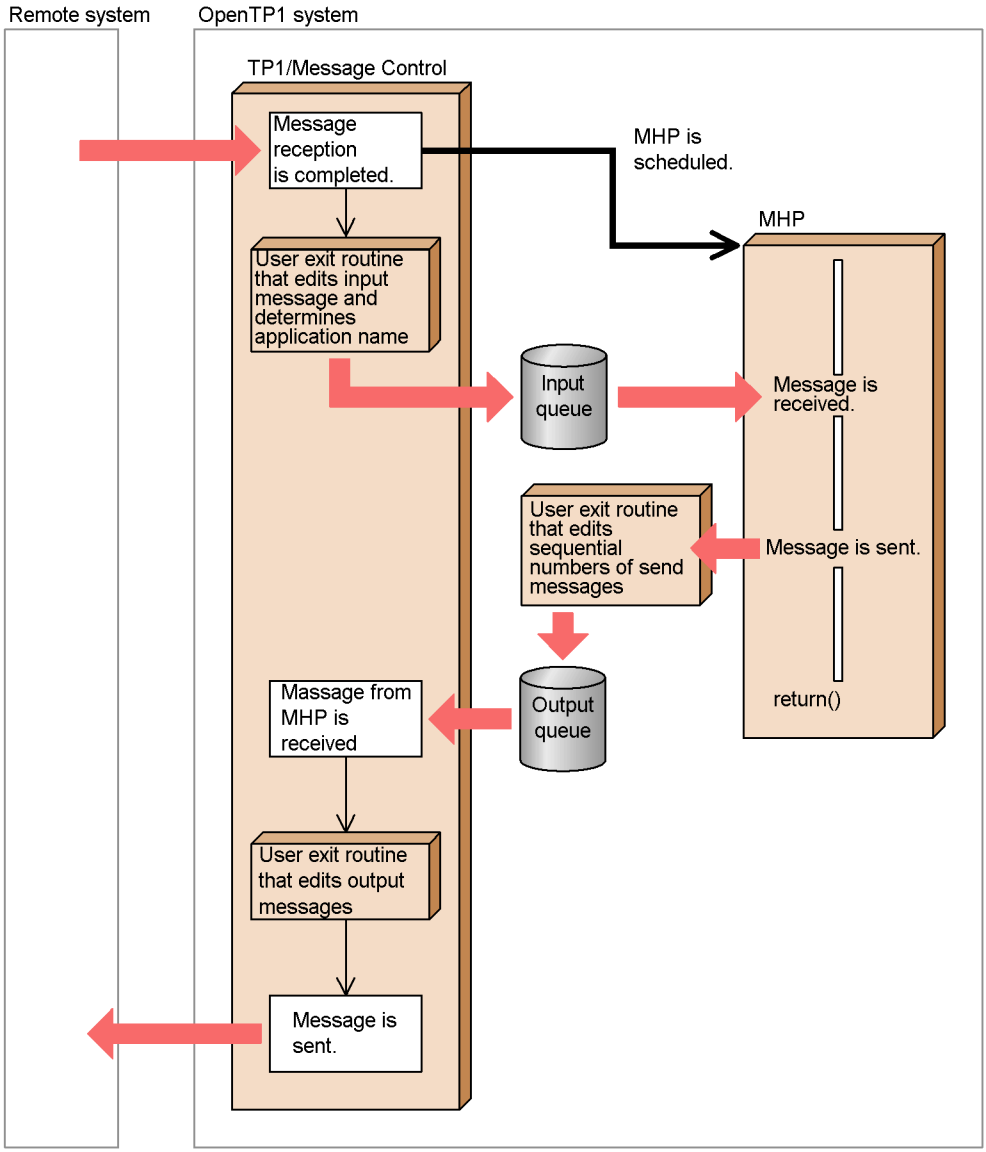
3.9 User exit routines

User exit routines are programs which help UAPs with message processing when OpenTP1 is in process of message exchange mode communication.

Either C or C++ is used for coding user exit routines. When the C is used, code the user exit routine in either the ANSI C format or the pre-ANSI K&R format (Classic C). When the C++ is used, code the user exit routine according to the C++ specification.

The figure below shows the relationship between message processing and user exit routines.

Figure 3-20: Positions of user exit routines



■ User exit routines available with OpenTP1

The table below lists user exit routines available with OpenTP1.

Table 3-15: User exit routines available with OpenTP1

Type of user exit routine	Processing that can be performed by user exit routine	Processing that is performed if user exit routine is not used
User exit routine that edits input messages/user exit routine that determines application name	• Edits received messages. • Determines the application name of the MHP which is to process the message.	The first up to 8 characters before the first space on the first segment are treated as the application name.
User exit routine that edits sequential number of send message	• Gives sequence numbers to segments to be sent.	The segments are given the sequence numbers specified in the function which is to send the message.
User exit routine that determines the inheriting timer-start message	• Can change the condition for activating timer-started applications after a rerun.	Whether the timer start message is inherited is determined according to the specification in the definition.
User exit routine that edits output message	• Edits the message to be output.	The message to be output is sent without editing.

The user exit routines listed in Table 3-15 vary in syntax depending on the communcation protocol product used with the MCF. You can use some user exit routines. There are also UOCs not included in this table that are specific to products that support particular communication protocols. For the syntax of user exit routines, see the applicable *OpenTP1 Protocol* manual.

In Table 3-15, user exit routines that determine the inheriting timer-start message do not depend on a communication protocol product. For the syntax of user exit routines, see the manual *OpenTP1 Programming Reference C Language*.

3.9.1 User exit routine that edits input message and application name determination

This user exit routine determines the application name of the MHP which processes input messages. When this user exit routine is incorporated, messages which OpenTP1 receives from other systems are passed to the user exit routine. After processing by the user exit routine ends, the message data is passed to the input queue. Then, the message data is transferred to the function for receiving MHP messages scheduled by OpenTP1.

When an MCF event informs and recovery processing is to be performed by the MCF event handling MHP, the user exit routine that edits input message and determines application name is not used.

For the format of the user exit routine that edits input message and determines application name, see the applicable *OpenTP1 Protocol* manual.

(1) Incorporation into OpenTP1

Specify the function address of the created user exit routine in the MCF main function (start function: `dc_mcf_svstart()`). The function address of the user exit routine that edits input message can be determined optionally. If the MCF main function is compiled and link-edited, the object file of the user exit routine is linked to the MHP execution form file and can be executed. For details on the MCF main function, see the manual *OpenTP1 Operation*.

3.9.2 User exit routine that determines the inheriting timer-start message

This user exit routine changes the environment for timer start when an error causes OpenTP1 to rerun after the timer started function `dc_mcf_execap()` is called or when the MCF service is rerun singly. This exit routine enables the following:

- Inherit or cancel the timer start specification
- Make the inherited timer start immediate start
- Change the name of the application to be started

(1) Incorporation into OpenTP1

Specify a function address of the created user exit routine in the MCF main function (`dc_mcf_svstart()`) for the application start service. Any function address may be specified. When the MCF main function is compiled and linked, the object file of the user exit routine is linked to the executable file of application start service and becomes ready to run. For details on the MCF main function for application start communication service, see the manual *OpenTP1 Operation*.

3.9.3 User exit routine that edits sequential number of send message

This user exit routine assigns serial numbers to send messages. It is started by specifying the function which sends messages from MHP.

This user exit routine is created as the `send_uoc()`. It is started when the first segment of the message send function is sent. Therefore, only the first segment can be edited by this user exit routine.

For the format of the user exit routine that edits sequential number of send message, see the applicable *OpenTP1 Protocol* manual.

(1) Incorporation into OpenTP1

Register the user exit routine as the function `dc_mcf_regster()` in the MHP main function.

3.9.4 User exit routine that edits output message

This user exit routine edits response messages or send-only messages. It needs to be positioned so that the send messages called by UAP are processed before they are actually sent to other systems.

For the format of the user exit routine that edits output message, see the applicable *OpenTP1 Protocol* manual.

(1) Incorporation into OpenTP1

Specify the function address name in the start function called by the MCF main function, in the same way as the user exit routine that edits input message and determines application name. The function address of the user exit routine that edits output message can be determined optionally. For details about MCF main function, see the manual *OpenTP1 Operation*.

3.10 MCF events

When messages are exchanged with OpenTP1, TP1/Message Control outputs the message which post various system information items of OpenTP1 to MHP. Such messages are called *MCF events*. If an error or failure reported during message exchange processing, what occurred in the system is indicated by an MCF event. There are two types of MCF events: error events such as errors and failures, and communication events dependent on the protocol such as establishment and release of a connection. The MHP which handles failures based on MCF events is called the *MHP for an MCF event*. Creating this MHP enables individual failure recovery processing.

An MCF event is passed to the input queue, and the MHP for an MCF event is started. At this time, the user exit routine that edits input message and determines application name is not used. An MCF event is never started as the result of a failure occurring in an MCF event.

The table below lists MCF events. Some MCF events which are not included in the table are reported as events specific to communication protocol supporting products. For MCF events specific to communication protocol supporting products, see the applicable *OpenTP1 Protocol* manual.

Table 3-16: MCF events

MCF event name	MCF event code	Cause of MCF event occurrence	Example of processing by MHP for an MCF event
MCF event that reports detection of an invalid application name	ERREVT1	The application name of the message was not found in the MCF application definition.	Posts that the application name was not found. For an inquiry message, a response message can be output.

MCF event name	MCF event code	Cause of MCF event occurrence	Example of processing by MHP for an MCF event
MCF event that reports discarding of a message	ERREVT2	The message in the input queue received with MCF or the message input to the input queue as a result of immediate startup of an application was discarded for any of the following reasons: • An error related to the input queue occurred. • MHP service, service group, or application was shut down. • MHP service, service group, or application is in secure state. • MHP terminated abnormally before the segment was passed to the function dc_mcf_ receive() of MHP. • There is no MHP service corresponding to the application name. • When MCF cannot activate SPP. • MHP is not running.	Posts that the message was discarded. For an inquiry message, a response message can be output.
MCF event that reports UAP abnormal termination	ERREVT3	MHP terminated abnormally or rolled back# after the segment was passed to the function dc_mcf_receive() invoked by MHP.	Reports that the UAP terminated abnormally or rolled back. For an inquiry message, a response message can be sent.
MCF event that reports discarding of a timer-start message	ERREVT4	The message input as a result of startup of the timer start application was discarded.	Posts that the message was discarded. In the case of an inquiry message, a response message can be output.
MCF event that reports discarding of an unprocessed send message	ERREVTA	The unprocessed message from a UAP was discarded for any of the following reasons: • Timeout occurred in residence time monitoring for unprocessed send message when MCF terminated normally. • The output queue was deleted by the mcftdlqle command or the function dc_mcf_tdlqle(). • The dcstop command was executed while a timer start request remained.	Posts that the unprocessed message was discarded. The unprocessed send message is saved in a file.
MCF event that reports a send error	SERREVT	A communication protocol error occurred during sending of a message.	Posts that the message could not be sent due to a failure in the communication protocol.

MCF event name	MCF event code	Cause of MCF event occurrence	Example of processing by MHP for an MCF event
MCF event that reports send completion	SCMPEVT	A message was sent normally to the remote system.	Posts that the message was sent normally to the remote system.
MCF event that reports an error	CERREVT (VERREVT)	A connection failure or logical terminal failure occurred with the communication management program. It does not report when automatic retry is specified.	Posts that a connection failure or logical terminal failure occurred.
MCF event that reports an status	COPNEVT (VOPNEVT)	Connection has been established.	Posts that connection has been established.
	CCLSEVT (VCLSEVT)	Connection has been released normally.	Posts that connection has been released.

Note

ERREVT1, ERREVT2, ERREVT3, ERREVT4, and ERREVTA represent error events.

SERREVT, SCMPEVT, CERREVT, COPNEVT, and CCLSEVT represent communication events.

#

Excludes cases in which r is specified for the recvmsg operand in the MCF application definition (mcfaalcap -g) or in which DCMCFRTRY or DCMCFRRTN is specified for action of the function dc_mcf_rollback().

- The application attribute of MHP for an MCF event

  The application attribute of the MHP for an MCF event is determined according to the cause of the MCF event occurrence. For the MHP for an MCF event, perform processing according to the determined application type.

  When starting the MHP for ERREVT1, ERREVT2, or ERREVT3, the application startup process is required. When this process is used, the MCF communication configuration definition needs to be created.

  If an MCF event occurs when two or more MHPs were started by the function dc_mcf_execap(), the type of the MHP for an MCF event is determined based on the type of the MHP that called the function dc_mcf_execap() first. When the function dc_mcf_execap() was called from SPP, the MCF event corresponding to the application startup process reports.

  The table below shows the relationship between the MHP for an MCF event and the application attribute.

Table 3-17: Relationship between MHPs for an MCF event and application attributes

Event code of MCF event	Application attribute of MHP for an MCF event
ERREVT1	The attribute is set according to the terminal type of the request source logical terminal. • *reply*-type logical terminals: ans • Logical terminals that are not the *reply* type: noans
ERREVT2	The application attribute of the MHP which caused MCFevent reporting is inherited as is.#
ERREVT3	
ERREVT4	
ERREVTA	The nonresponse (noans) type is set.
SERREVT	
SCMPEVT	
CERREVT	
VERREVT	
COPNEVT	
CCLSEVT	
VCLSEVT	

#

If the MHP has the nontransaction attribute, the application attribute is not inherited even after abnormal termination; instead, the specification for the MHP for an MCF event is observed.

- Relationship between communication protocol products and reported MCF events

  The following tables show the relationships between communication protocol products and reported MCF events.

  Table 3-18: Relationship between communication protocol products and reported MCF events (1/5)

MCF EVENT	Communication protocol product		
	TP1/NET/User Agent	TP1/NET/OSI-TP	TP1/NET/TCP/IP
ERREVT1	Y	Y	Y

MCF EVENT	Communication protocol product		
	TP1/NET/User Agent	TP1/NET/OSI-TP	TP1/NET/TCP/IP
ERREVT2	Y	Y	Y
ERREVT3	Y	Y	Y
ERREVT4	Y	Y	Y
ERREVTA	Y	Y	Y
SERREVT	N	N	N
SCMPEVT	N	N	Y
CERREVT	Y	Y	Y
COPNEVT	Y	Y	Y
CCLSEVT	Y	Y	Y
VERREVT	N	N	N
VOPNEVT	N	N	N
VCLSEVT	N	N	N

Legend:

Y: The event is reported by the communication protocol product.

N: The event is not reported by the communication protocol product.

Table 3-19: Relationship between communication protocol products and reported MCF events (2/5)

MCF EVENT	Communication protocol product		
	TP1/NET/XMAP3	TP1/NET/HNA-560/20	TP1/NET/HNA-560/20 DTS
ERREVT1	Y	Y	Y
ERREVT2	Y	Y	Y
ERREVT3	Y	Y	Y
ERREVT4	Y	Y	Y
ERREVTA	Y	Y	Y
SERREVT	Y#	N	N

MCF EVENT	Communication protocol product		
	TP1/NET/XMAP3	TP1/NET/HNA-560/20	TP1/NET/HNA-560/20 DTS
SCMPEVT	Y#	N	N
CERREVT	N	Y	Y
COPNEVT	N	Y	Y
CCLSEVT	N	N	N
VERREVT	Y	Y	Y
VOPNEVT	Y	Y	Y
VCLSEVT	Y	Y	N

Legend:

Y: The event is reported by the communication protocol product.

N: The event is not reported by the communication protocol product.

#

SERREVT and SCMPEVT are reported only when the print facility is used.

Table 3-20: Relationship between communication protocol products and reported MCF events (3/5)

MCF EVENT	Communication protocol product			
	TP1/NET/ OSAS-NIF	TP1/NET/ HNA-NIF	TP1/NET/HSC (1)	TP1/NET/HSC (2)
ERREVT1	Y	Y	Y	Y
ERREVT2	Y	Y	Y	Y
ERREVT3	Y	Y	Y	Y
ERREVT4	N	N	Y	Y
ERREVTA	Y	Y	Y	Y
SERREVT	N	N	Y	Y#
SCMPEVT	N	N	Y	Y#
CERREVT	Y	Y	Y	Y

MCF EVENT	Communication protocol product			
	TP1/NET/ OSAS-NIF	TP1/NET/ HNA-NIF	TP1/NET/HSC (1)	TP1/NET/HSC (2)
COPNEVT	Y	Y	Y	Y
CCLSEVT	Y	Y	Y	Y
VERREVT	N	N	N	N
VOPNEVT	N	N	N	N
VCLSEVT	N	N	N	N

Legend:

Y: The event is reported by the communication protocol product.

N: The event is not reported by the communication protocol product.

#

TP1/NET/HSC reports SERREVT and SCMPEVT only in the asynchronous mode.

Table 3-21: Relationship between communication protocol products and reported MCF events (4/5)

MCF EVENT	Communication protocol product		
	TP1/NET/HDLC	TP1/NET/X25	TP1/NET/ X25-Extended
ERREVT1	Y	Y	Y
ERREVT2	Y	Y	Y
ERREVT3	Y	Y	Y
ERREVT4	Y	Y	Y
ERREVTA	Y	Y	Y
SERREVT	N	N	N
SCMPEVT	Y	N	Y
CERREVT	Y	Y	Y
COPNEVT	Y	Y	Y
CCLSEVT	Y	Y	Y
VERREVT	N	N	N

MCF EVENT	Communication protocol product		
	TP1/NET/HDLC	TP1/NET/X25	TP1/NET/ X25-Extended
VOPNEVT	N	N	N
VCLSEVT	N	N	N

Legend:

Y: The event is reported by the communication protocol product.

N: The event is not reported by the communication protocol product.

Table 3-22: Relationship between communication protocol products and reported MCF events (5/5)

MCF EVENT	Communication protocol product			
	TP1/NET/ SLU-TypeP1	TP1/NET/ SLU-TypeP2	TP1/NET/ NCSB	TP1/NET/UDP
ERREVT1	Y	Y	Y	Y
ERREVT2	Y	Y	Y	Y
ERREVT3	Y	Y	Y	Y
ERREVT4	Y	Y	Y	Y
ERREVTA	Y	Y	Y	Y
SERREVT	N	N	N	N
SCMPEVT	N	N	N	N
CERREVT	Y	Y	Y	Y
COPNEVT	Y	Y	Y	Y
CCLSEVT	Y	Y	Y	Y
VERREVT	N	N	N	N
VOPNEVT	N	N	N	N
VCLSEVT	N	N	N	N

Legend:

Y: The event is reported by the communication protocol product.

N: The event is not reported by the communication protocol product.

3.10.1 MCF event that reports detection of an invalid application name (ERREVT1)

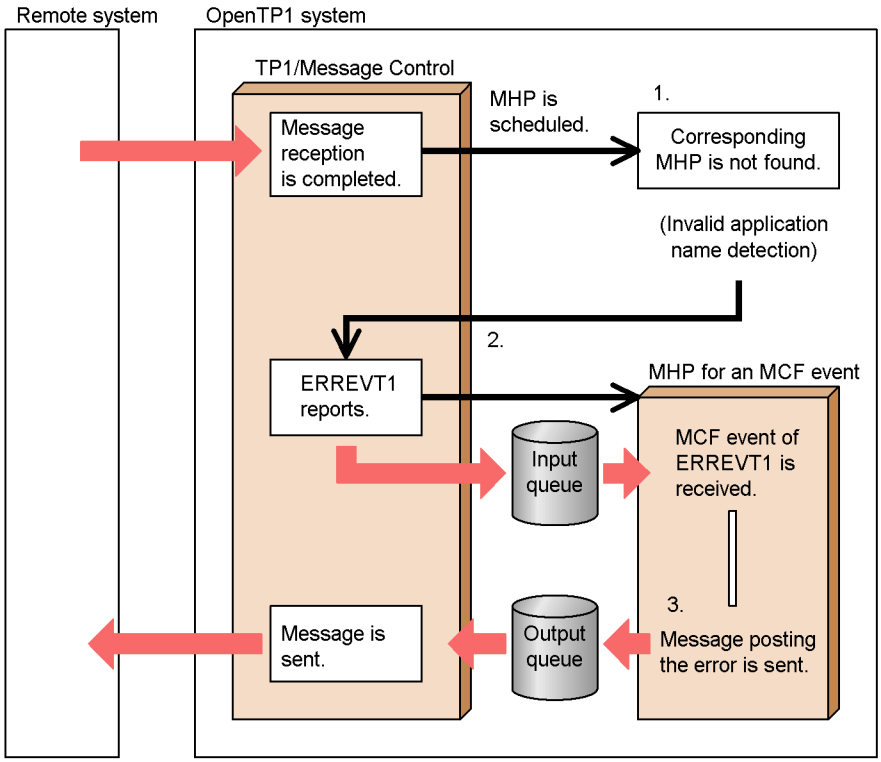
ERREVT1 is reported if the application name specified in the received message is invalid for any of the following reasons:

- The format of application name is invalid.
- The specified application name is not found in the MCF application definition.

With the MHP for ERREVT1, send the send-only message posting that the application name was not found in the local node or take similar measures. At this time, send a response message or send only-message from the MHP for an MCF event depending on the type of the logical terminal or UAP.

The figure below shows the outline of ERREVT1.

Figure 3-21: Outline of ERREVT1



1. The MCF which received the message attempted to schedule the MHP corresponding to the application name, but the MHP was not found.
2. Control returns to the MCF, ERREVT1 reports, and the MHP for ERREVT1 is

scheduled.

3. The MHP for an MCF event sends the send only message posting that there is no MHP corresponding to the message.

3.10.2 MCF event that reports discarding of a message (ERREVT2)

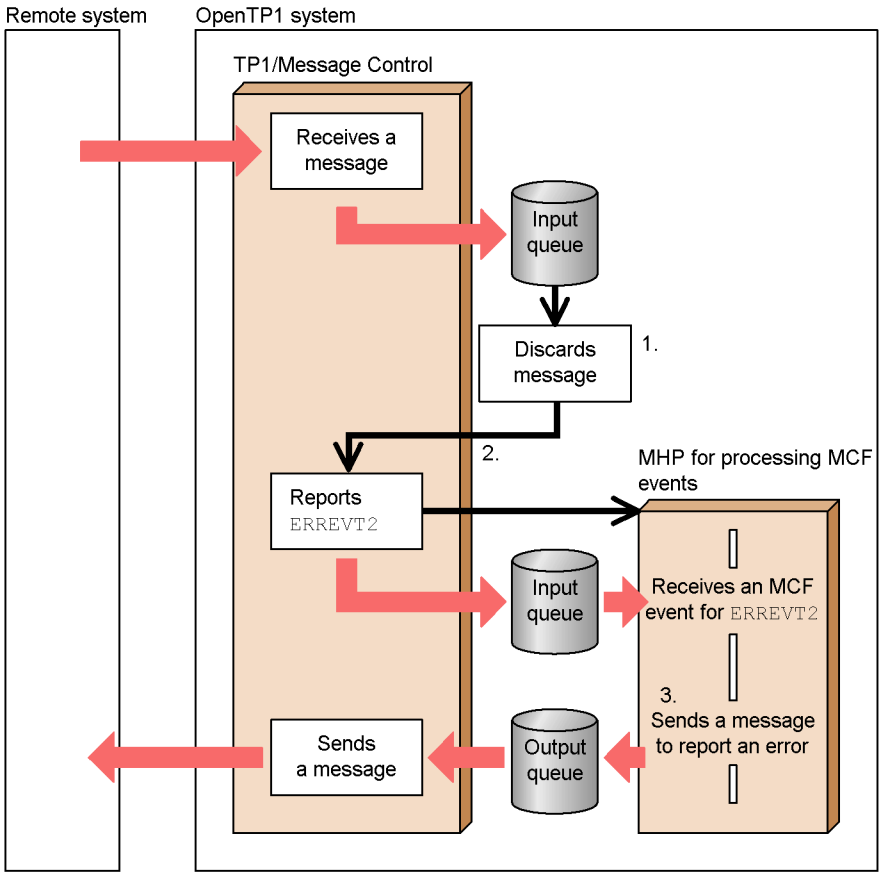
ERREVT2 is reported when the received message was discarded for any of the following reasons. ERREVT2 is also reported when a communication event for which `errevt=yes` (report error event at communication event failure) is specified in the `-n` option of the application attribute definition `mcfaalcap` encounters a failure for any of the following reasons:

- An input queue error occurred.
- An application, service, or service group was blocked or is in secure state.
- MHP terminated abnormally before the first segment is received.
- There is no MHP service function corresponding to the application name.
- SPPs cannot be started by the MCF when the remote MCF service is used.
- Messages remain in the input queue due to schedule blocking of service groups at termination of OpenTP1.

With the MHP for ERREVT2, reference the contents of ERREVT2 and send the message posting that processing was not possible in the local node or take similar measures. At this time, send a response message or send only-message from the MHP for an MCF event depending on the type of the logical terminal or UAP.

The figure below shows the outline of ERREVT2.

Figure 3-22: Outline of ERREVT2



1. The received message was discarded from the input queue for some reason.
2. Control returns to MCF, ERREVT2 reports, and the MHP for ERREVT2 is scheduled.
3. The send-only message posting the message resend request and so forth is sent from the MHP for an MCF event to the other system which sent the message.

3.10.3 MCF event that reports UAP abnormal termination (ERREVT3)

ERREVT3 is reported in the following case. ERREVT3 is also reported when a communication event for which `errevt=yes` (report error event at communication event failure) is specified in the `-n` option of the application attribute definition `mcfaalcap` encounters a failure for any of the following reasons:

- MHP terminated abnormally after the first segment was received with the function `dc_mcf_receive()`.

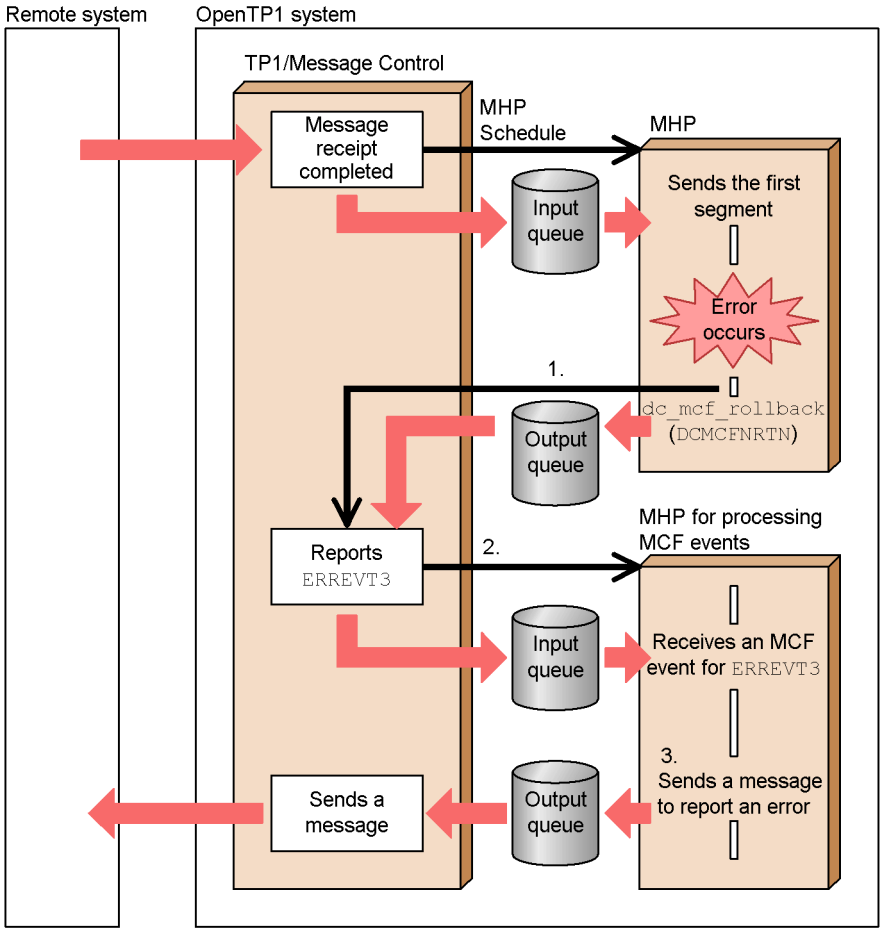
- A failure occurred at termination of an application.

This event is reported when DCMCFNRTN is set for the flag of the function `dc_mcf_rollback()` called by the MHP.

With the MHP for ERREVT3, reference the contents of ERREVT3 and send the message posting that the UAP of the local node terminated abnormally, using the application name as the key, or take similar measures. At this time, send a response message or send-only message from the MHP for an MCF event depending on the type of the logical terminal or UAP.

The figure below shows the outline of ERREVT3.

Figure 3-23: Outline of ERREVT3



1. When retry is not set for rollback, if an error occurs in the processing with the

MHP which received the message, control returns to MCF via the output queue.

2. ERREVT3 reports based on the information sent from the MHP which caused the error.
3. ERREVT3 schedules the MHP for an MCF event via the input queue. This MHP sends the message posting that an error occurred in the UAP of the another system, to the other system which sent the message.

3.10.4 MCF event that reports discarding of a timer-start message (ERREVT4)

ERREVT4 is reported when the message was discarded for the following reason:

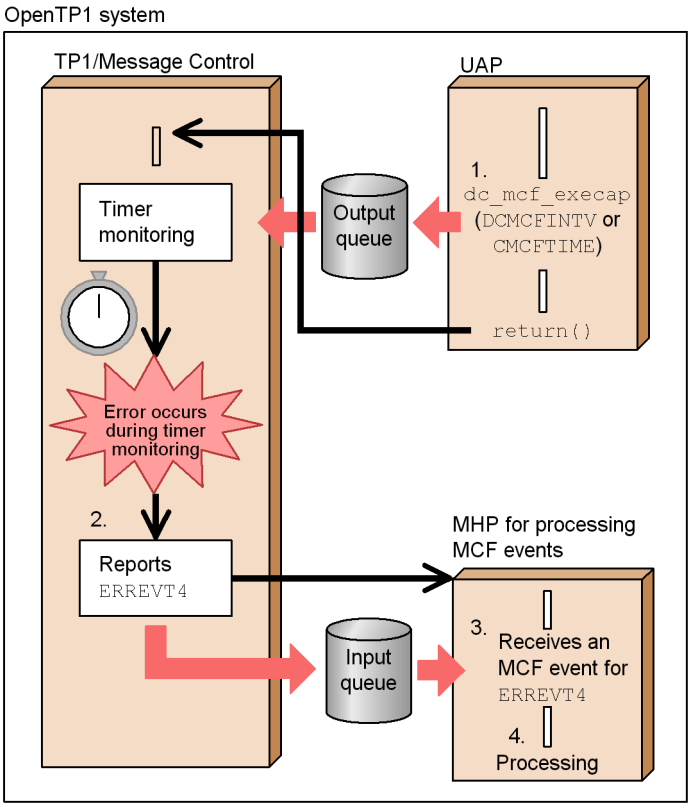
- The message was discarded because an error occurred during timer monitoring by the MCF after the application program start function (the function `dc_mcf_execap()`) with timer start was called from a UAP.

(1) Flow up to ERREVT4

If MHP calls the function `dc_mcf_execap()` of timer start, MCF fetches messages from the output queue and performs timer monitoring. If a timer monitoring error, scheduling error, or the like reports during the wait time before writing in the input queue, ERREVT4 occurs. After that, the MHP to process ERREVT4 is started.

The figure below shows the outline of ERREVT4.

Figure 3-24: Outline of ERREVT4



1. The function `dc_mcf_execap()` of timer start is called. If the transaction commits it, MCF starts timer monitoring.
2. If a failure reports during timer monitoring by MCF, ERREVT4 reports.
3. ERREVT4 schedules the MHP for ERREVT4 via the input queue.
4. The MHP for an MCF event analyzes and processes ERREVT4.

3.10.5 MCF event that reports discarding of an unprocessed send message (ERREVTA)

ERREVTA is reported in the following cases:

- The messages remaining in the output queue were discarded because timeout occurred in residence time monitoring for unprocessed send messages after the normal termination command for OpenTP1 (the `dcstop` command) was executed.
- The output queue containing unprocessed send messages was deleted with the

mcftdlqle command or the function dc_mcf_tdlqle() while OpenTP1 was running.

- The timer-start function dc_mcf_execap() was called, and the command for terminating OpenTP1 normally (dcstop command) was executed during timer monitoring.

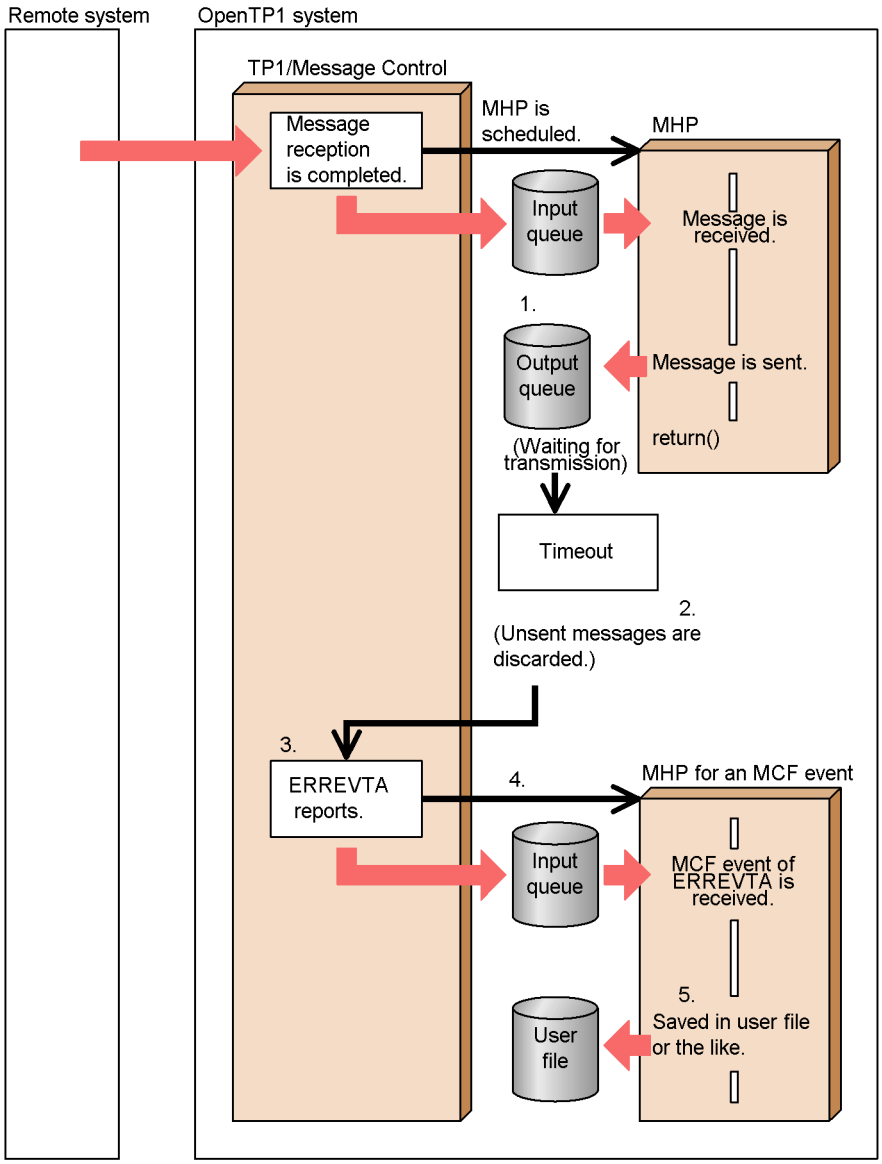
(1) Flow up to ERREVTA

When MHP terminates normally, the messages sent to the output queue are output. When OpenTP1 is to be terminated normally in transmission wait state, MCF waits for the termination until the send messages in the output queue have been sent. At this time, if the messages cannot be sent due to a failure in the destination system, timeout occurs and the send messages are discarded. ERREVTA reports to post that the messages have been discarded. The period of time causing timeout is specified for the mtim operand of the timer definition mcfttim of the MCF communication configuration definition. Timer monitoring is performed based on this operand value.

Note that the timer start request message issued by the function dc_mcf_execap() is not included in the monitoring of the remaining time for unprocessed send messages. Consequently, when the normal termination command (dcstop command) for OpenTP1 is executed, the timer start request message is discarded immediately and ERREVTA is reported.

The figure below shows the outline of ERREVTA.

Figure 3-25: Outline of ERREVTA



1. The normal termination command (`dcstop` command) for OpenTP1 is executed. Any timer start request message remaining at this time is discarded, and MCF reports ERREVTA.
2. The message that was processed normally by MHP is stored in the output queue.

3. Output messages are discarded because timeout occurred for the send messages in the output queue.
4. ERREVTA reports from MCF.
5. The MHP for ERREVTA is scheduled.
6. The message information is saved in a user file or the like.

3.10.6 MCF event that reports a send error (SERREVT)

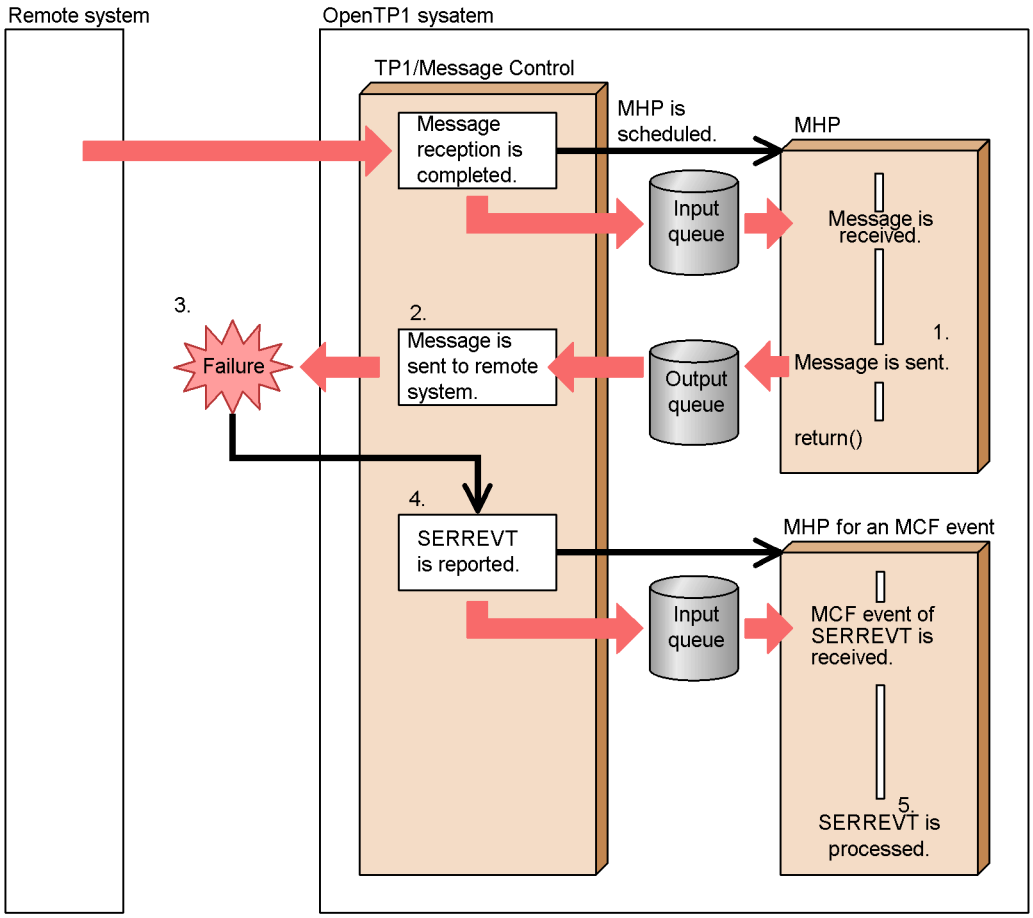
SERREVT is reported when a failure occurs in the communication protocol while the MCF is sending a message to a remote system after the UAP that sent the message normally has terminated processing. Be referencing this event, you can confirm that a failure occurred in the communication protocol even if asynchronous message send processing was used (function `dc_mcf_send()` and function `dc_mcf_reply()`).

The MHP for an MCF event of SERREVT is a nonresponse-type (`noans` type) MHP.

SERREVT is not reported if you terminate OpenTP1 before events are written to the input queue.

The figure below shows the outline of SERREVT.

Figure 3-26: Outline of SERREVT



1. *Report event* is set in the arguments of the function `dc_mcf_send()` or function `dc_mcf_reply()` and the message is sent.
2. The UAP terminates normally. The MCF that received a send request from the UAP sends a message to the remote system.
3. A failure occurs in the communication protocol.
4. Control returns to the MCF. SERREVT is reported and the MHP for an MCF event is scheduled.
5. The MHP for an MCF event processes SERREVT in accordance with the details reported by SERREVT.

3.10.7 MCF event that reports send completion (SCMPEVT)

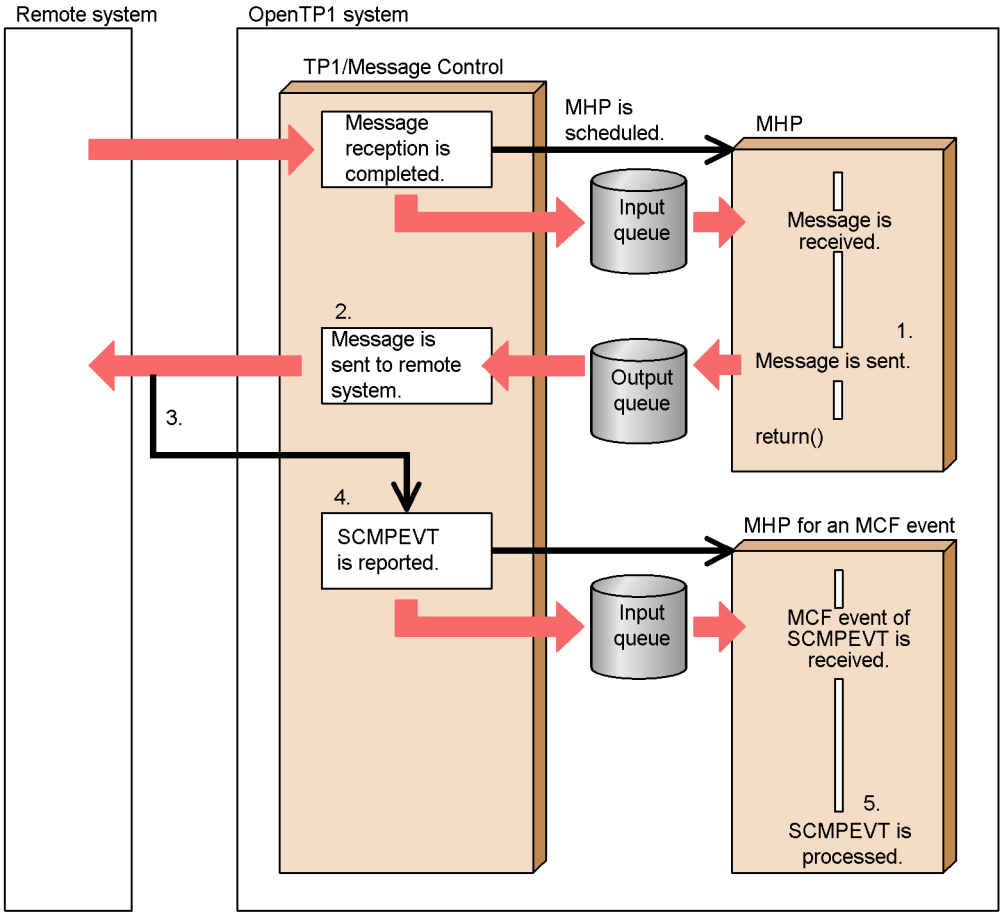
SCMPEVT is reported by the MCF when a message was sent normally. This event indicates that an asynchronous message was sent (function dc_mcf_send() and function dc_mcf_reply()) normally to the remote system.

The MHP for an MCF event of SCMPEVT can start processing for synchronization with send completion. At this time, the MHP for an MCF event is a nonresponse-type (noans type) MHP.

SCMPEVT is not reported if you terminate OpenTP1 before events are written to the input queue.

The figure below shows the outline of SCMPEVT.

Figure 3-27: Outline of SERREVT



1. *Report event* is set in the arguments of the function `dc_mcf_send()` or function `dc_mcf_reply()` and the message is sent.
2. The MCF that received a send request from the UAP sends a message to the remote system.
3. The message is sent normally to the remote system.
4. Control returns to the MCF. `SCMPEVT` is reported and the MHP for an MCF event is scheduled.
5. The MHP for an MCF event processes `SCMPEVT` in accordance with the details reported by `SCMPEVT`.

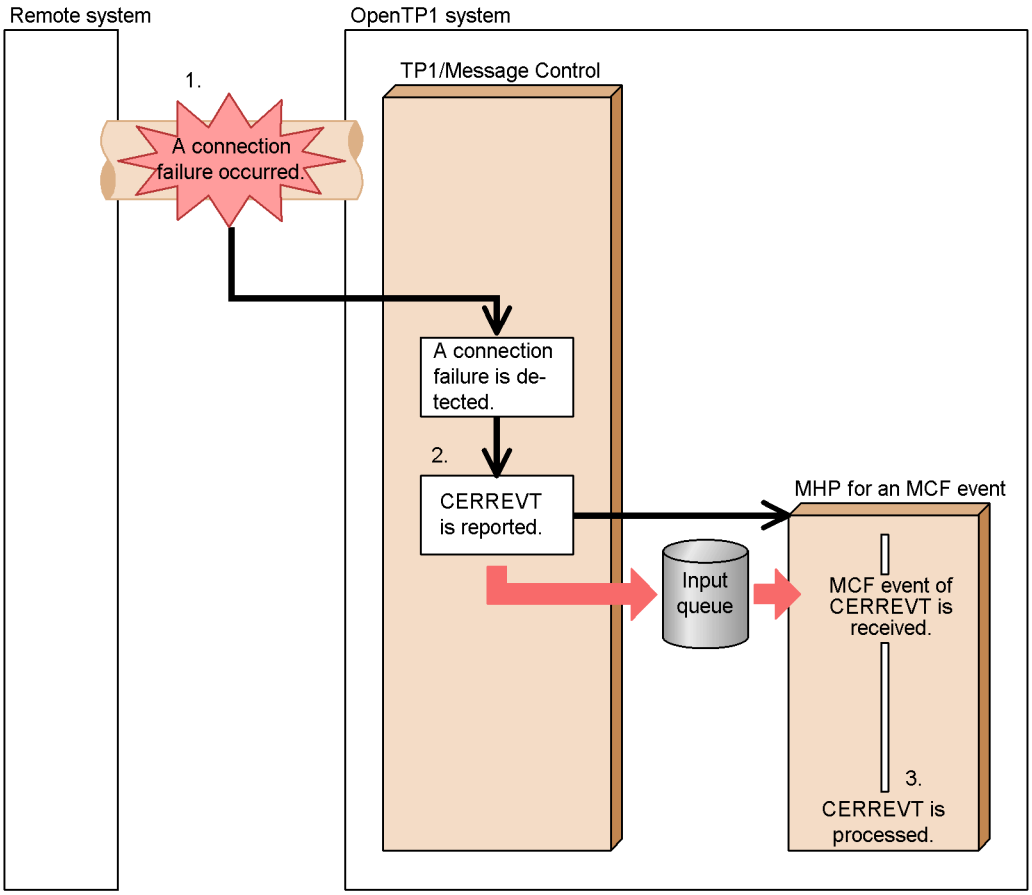
3.10.8 MCF event that reports an error (CERREVT, VERREVT)

`CERREVT` (`VERREVT`) is reported if a connection failure or logical termination failure occurred with the communication management program. `CERREVT` (`VERREVT`) is not reported when retry of establishing connection is specified in the protocol specific definition of MCF communication configuration definition.

The way of reporting a connection failure depends on the protocol supporting product. For details on the format of `CERREVT`, see the applicable *OpenTP1 Protocol* manual.

The figure below shows the outline of `CERREVT` (`VERREVT`).

Figure 3-28: Outline of CERREVT (VERREVT)



1. A connection failure occurred during communication with the remote system.
2. CERREVT (VERREVT) is reported and the MHP for an MCF event is scheduled if retry of establishing connection is not specified or the retry count exceeds the specified value.
3. The MHP for an MCF event performs proper processing for CERREVT (VERREVT).

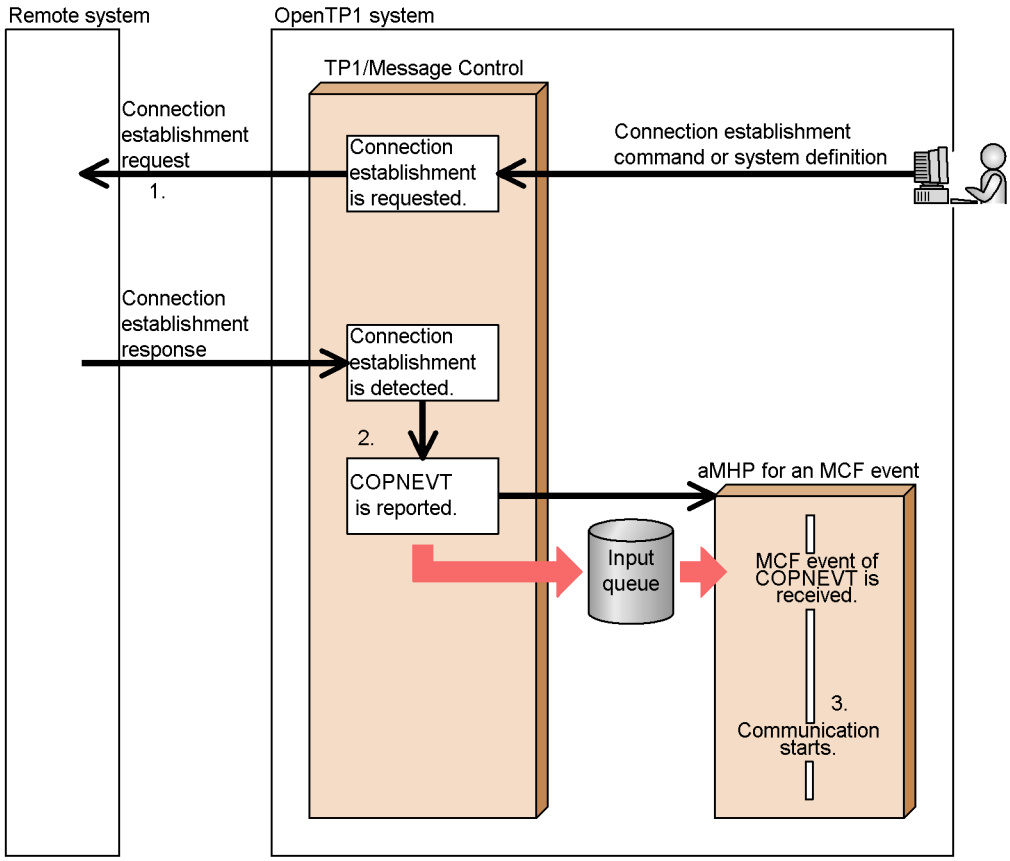
3.10.9 MCF event that reports establishing a connection (COPNEVT, VOPNEVT)

COPNEVT (VOPNEVT) is reported when a connection has been established between the remote system and the MCF or communication management program. The MHP can recognize that a connection has been established by receiving COPNEVT (VOPNEVT).

The way of reporting connection establishment depends on the protocol supporting product. For details on the format of COPNEVT (VOPNEVT), see the applicable *OpenTP1 Protocol* manual.

The figure below shows the outline of COPNEVT (VOPNEVT).

Figure 3-29: Outline of COPNEVT (VOPNEVT)



1. Connection establishment is requested from the local OpenTP1 system or the remote system. In this example, the local OpenTP1 system requests connection establishment. Depending on the protocol supporting product, there may be no response.
2. When a connection has been established, COPNEVT (VOPNEVT) is reported and the MHP for an MCF event is scheduled.
3. The MHP for an MCF event performs proper processing for COPNEVT (VOPNEVT).

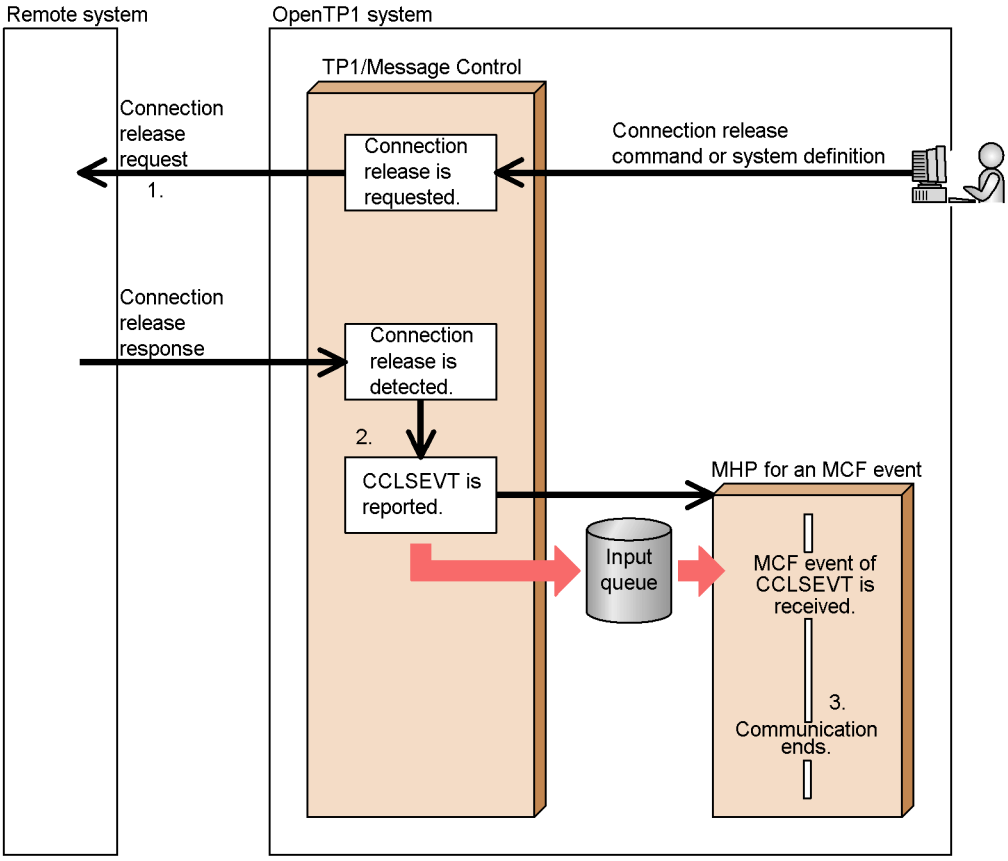
3.10.10 MCF event that reports releasing a connection (CCLSEVT, VCLSEVT)

CCLSEVT (VCLSEVT) is reported when the connection between the remote system and the MCF or communication management program has been released . The MHP can recognize that a connection has been released by receiving CCLSEVT (VCLSEVT).

The way of reporting connection release depends on the protocol supporting product. For details on the format of CCLSEVT (VCLSEVT), see the applicable *OpenTP1 Protocol* manual.

The figure below shows the outline of CCLSEVT (VCLSEVT).

Figure 3-30: Outline of CCLSEVT (VCLSEVT)



1. Connection release is requested from the local OpenTP1 system or the remote system. In this example, the local OpenTP1 system requests connection release. Depending on the protocol supporting product, there may be no response.

2. When the connection has been released, CCLSEVT (VCLSEVT) is reported and the MHP for an MCF event is scheduled.
3. The MHP for an MCF event performs proper processing for CCLSEVT (VCLSEVT).

3.10.11 Message format for MCF events

Logical messages passed as MCF events comprise MCF event information and unprocessed messages. Unprocessed messages vary with reported MCF events as follows:

ERREVT1

Message which did not allow the application to be identified

ERREVT2

Message which failed to be passed to the target application because of application shutdown or other condition

ERREVT3

Message received by an MHP (application) which terminated abnormally

ERREVT4

Message which was about to be passed to an MHP when the application program was timer-started

ERREVTA

Message which remained in the output queue

If one of the following MCF events is reported, only MCF event information is passed. There is no unprocessed message.

- SERREVT
- SCMPEVT
- CERREVT (VERREVT)
- COPNEVT (VOPNEVT)
- CCLSEVT (VCLSEVT)

(1) MCF event message structure

When an MHP is to receive an MCF event, it uses the function for receiving ordinary messages (dc_mcf_receive()).

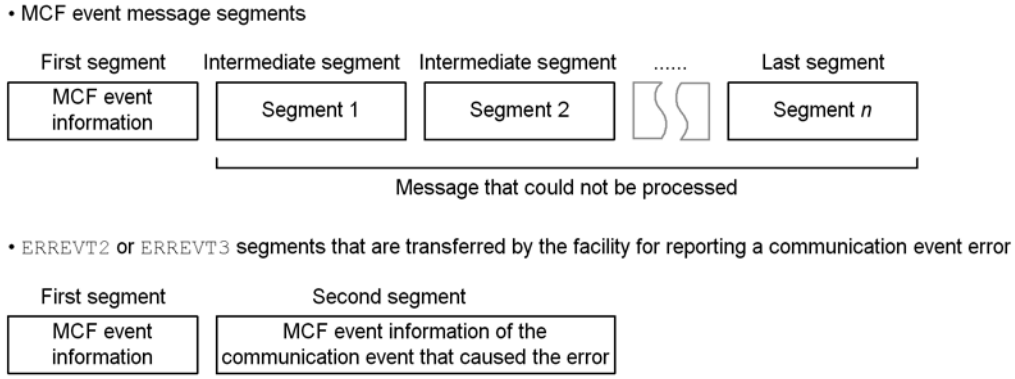
The MCF event is passed to the MHP for an MCF event as a logical message consisting of multiple segments. The first segment has MCF event information and the second and subsequent segments carry the segments of the unprocessed message. The first

segment of the original message is on the second segment of the MCF event message and the *n*-th segment of the original message is on the (*n* + 1)-th segments of the MCF event message.

ERREVT2 or ERREVT3 is passed to the MHP for an MCF event that is started by the error event notification facility (when `errevt=yes` is specified in the `-n` option of the application attribute definition `mcfaalcap`) when a failure occurs in a communication event. MCF event information is set in the first segment and MCF event information about the communication event that failed is set in the second segment.

The figure below shows the segment layout of a logical message passed as an MCF event.

Figure 3-31: Segments of logical message passed as MCF event



(2) Format of data reported

MCF event information is reported according to the high-level language (C or COBOL) in which the MHP for an MCF event is written.

For MHPs written in C, MCF event information can be received with a structure. The structure is defined in the header file <`dcmcf.h`>. Include <`dcmcf.h`> with the `#include` statement for the MHP which handles MCF event information. For some communication events, the structure may be defined in the header file for each communication protocol supporting product.

For MHPs written in COBOL, MCF event information can be received with a segment list. Desired data can be fetched from any byte position of the segment.

The data format of MCF event information varies with the communication protocol supporting product (TP1/NET/*xxxx*). For details of the data format of the following MCF event information, see the applicable *OpenTP1 Protocol* manual:

- ERREVT1
- ERREVT2

- ERREVT3
- ERREVTA
- SERREVT
- SCMPEVT
- CERREVT (VERREVT)
- COPNEVT (VOPNEVT)
- CCLSEVT (VCLSEVT)

For details on the data format of ERREVT4 MCF event information, see the applicable *OpenTP1 Programming Reference* manual.

3.11 MCF processes used by application programs

This section explains MCF processes used by UAPs. When UAPs exchange messages, the following MCF service processes are used:

- MCF communication process

  This system process is used when the local OpenTP1 system communicates with the remote system.

- Application startup process

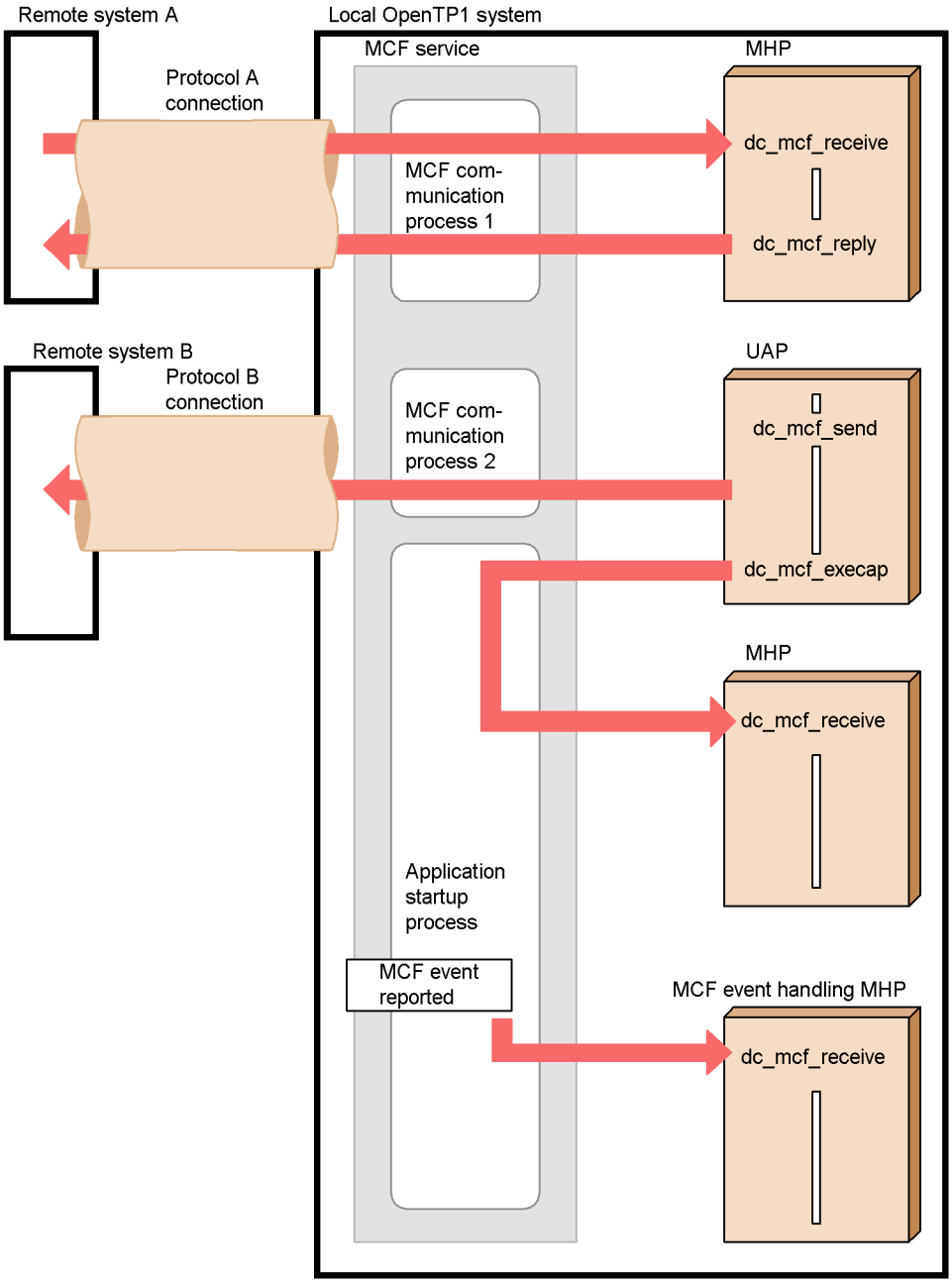
  This system process is used when the OpenTP1 system exchanges messages internally.

The reasons why MCF processes are classified as shown above are:

1. The load of processes used for MCF communication can be distributed because the system processes are separated for external and internal communication.
2. The process for internal communication is not effected while MCF communication processes cannot be used due to a communication protocol error.

The figure below shows the outline of MCF processes used by UAPs.

Figure 3-32: Outline of MCF processes used by UAPs



3.11.1 Types of MCF process

This subsection explains the MCF processes (MCF communication process and application startup process).

(1) MCF communication process

An MCF communication process is used when the local OpenTP1 system communicates with the remote system. This process is used when a UAP communicates with the remote system by using the following functions:

- Receiving message (the function `dc_mcf_receive()`)
- Sending message (the function `dc_mcf_send()`)
- Sending response message (the function `dc_mcf_reply()`)
- Receiving synchronous message (the function `dc_mcf_recvsync()`)
- Sending synchronous message (the function `dc_mcf_sendsync()`)
- Exchanging synchronous message (the function `dc_mcf_sendrecv()`)
- Resending message (the function `dc_mcf_resend()`)

Create an MCF communication process for each communication protocol supporting product. When one OpenTP1 system communicates with a remote system via multiple communication protocols, define an MCF communication process for each communication protocol supporting product.

(2) Application startup process

This system process is used when the OpenTP1 system passes messages to an internal MHP.

The application startup process is used when the following facilities are executed:

- Starting application program (the function `dc_mcf_execap()`)
- MHP rollback (the function `dc_mcf_rollback()`) with retry specification
- When reported MCF error events (`ERREVT`*x*) are used in a job.
- When an MHP is activated with the `mcfuevt` command.

The application startup process is not used to communicate with other systems (does not depend on the communication protocol). In general, define one application starting process for each node.

3.11.2 Files for using MCF processes

Prepare the following files to use the MCF service:

- Definition object file

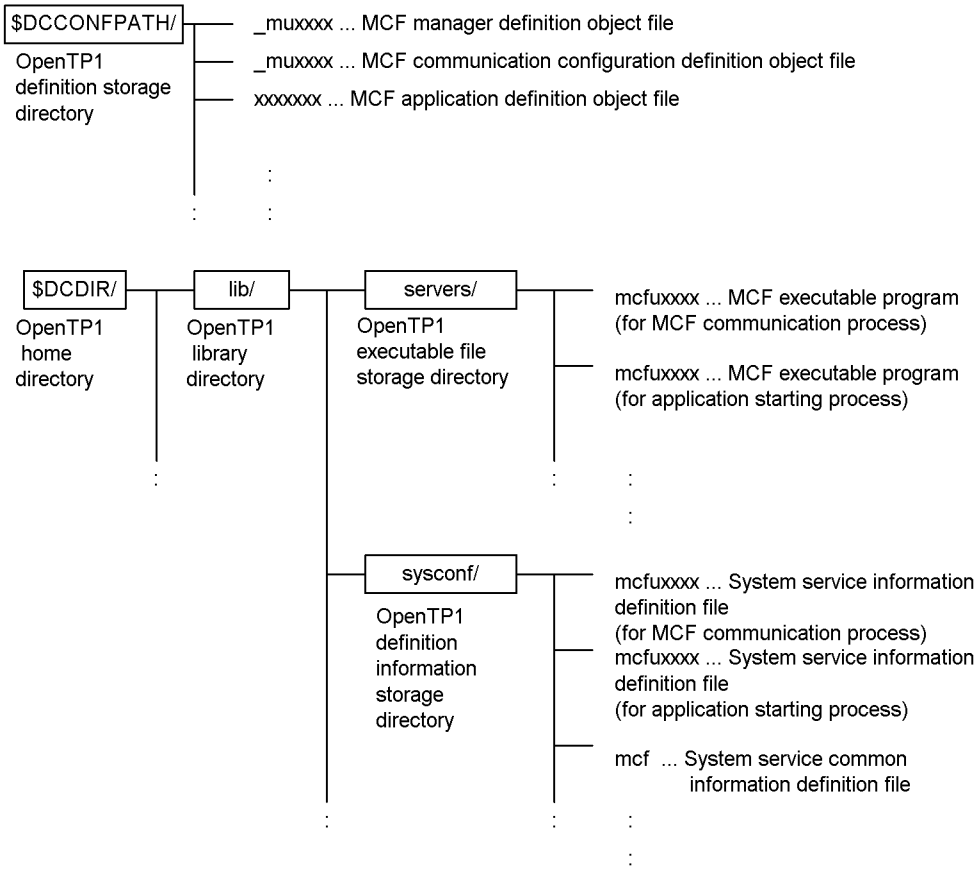
- MCF executable program
- System service information definition file

For MCF communication processes, the contents of definition and syntax of the command for creating files vary with the communication protocol supporting product. For definitions relating to MCF communication process and definition command utilities, see the applicable *OpenTP1 Protocol* manual.

For application startup processes, the files can be created independent of the communication protocol. For definitions relating to application starting process and definition command utilities, see the manuals *OpenTP1 System Definition* and *OpenTP1 Operation*.

The figure below shows the configuration of directories for storing the files needed to use the MCF service.

Figure 3-33: Configuration of directories for storing files needed to use MCF service



Chapter
4. Facilities for User Data

This chapter explains the following:

Facilities for using the OpenTP1 file services (TP1/FS/Direct Access, TP1/FS/Table Access)

IST service (TP1/Shared Table Access)

Facilities for using ISAM files

Accessing database management systems

Facility for locking files and other resources

The facilities are explained using C-language function names. For each function, the name of the equivalent COBOL-language API function is indicated in brackets [ ] when the function appears first in this chapter. After that, only the C-language function name is written. If the C-language function has no COBOL counterpart API function, brackets are not written.

This chapter contains the following sections:

4.1 DAM file service (TP1/FS/Direct Access)
4.2 TAM file service (TP1/FS/Table Access)
4.3 IST service (TP1/Shared Table Access)
4.4 ISAM file service (ISAM, ISAM/B)
4.5 Accessing database management systems
4.6 Lock for resources
4.7 Responses to the occurrence of deadlocks

4.1 DAM file service (TP1/FS/Direct Access)

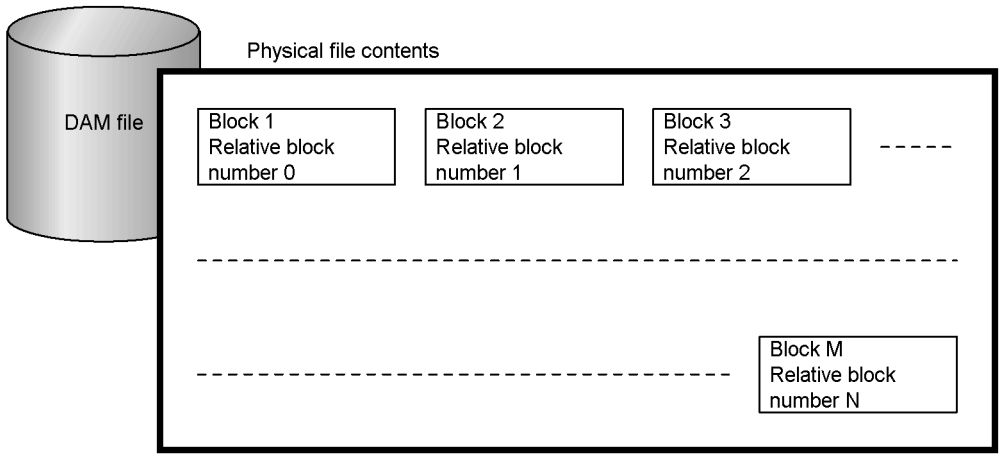
This section explains DAM files which can be used as user files dedicated to OpenTP1. Before DAM files can be used, TP1/FS/Direct Access must be installed in the system and the basic OpenTP1 facility must be TP1/Server Base. The DAM file service is unavailable with TP1/LiNK.

4.1.1 DAM file configuration

A DAM file comprises data items. Each data item is called a *block*. Specify one block of the DAM file in bytes (sector length x *n* - 8). Calculate the sector length with the value of the disk unit to be used.

The figure below shows the DAM file configuration.

Figure 4-1: DAM file configuration



4.1.2 Physical files and logical files

This subsection explains DAM file names to be specified in functions which are issued from UAPs.

(1) Physical files

To create DAM files, areas to be used as the DAM files must be allocated with `damload` command or a UAP that handles offline work. The name to be used for this allocation must be the pathname of the character type special file allocated as the OpenTP1 file system. The file identified by this pathname is referred to as the *physical file* and the file name is referred to as the *physical file name*. The physical file name is used when creating initial data in the allocated DAM file and updating the contents of the file in an offline environment.

(2) Logical files

When DAM files are used from SUPs, SPPs, or MHPs in online mode, the logical files given names in the system definition are accessed. These files are referred to as *logical files* and their names are referred to as *logical file names*. Logical and physical file names are in one-to-one correspondence. This correspondence is specified in the DAM service definition.

(3) Notes on access from UAPs

Since logical and physical files are defined at each node, DAM files are managed at each node independently of other nodes. This means that UAPs cannot use DAM files located at other nodes. To use DAM files, a UAP process in the node (machine) must access the files with the file names defined in the node.

4.1.3 Outline of access to DAM files

This subsection explains how to access DAM files. There are two types of DAM files:

- Recoverable DAM file

  Block input/output is synchronized with UAP transaction processing.

- Unrecoverable DAM file

  Block input/output is independent of transaction processing.

Hereafter, the explanations of the facilities specific to recoverable DAM files are given under the heading "Recoverable DAM files". For details on unrecoverable DAM files, see *4.1.8 Access to unrecoverable DAM files*.

(1) Procedure for access to DAM files

When a UAP uses a DAM file, the UAP accesses the file with the following procedure:

1. Open the file.
2. Perform one of the following processes:

   Data input (reference), data input and updating, and data output.
3. Close the file.

(2) Notes on opening DAM files

The function to open a DAM file must be called from each UAP that will use the DAM file. Even if more than one UAP belongs to one global transaction, each of the UAPs must call the function to open the DAM file.

- Recoverable DAM files

  Logical files can be opened both inside and outside the transaction. In the following cases, however, only block locking can be specified for logical files:

- The logical file must be opened before the transaction starts.
- Resource locking is specified for each global transaction.

(3) Notes on receiving blocks from DAM files

DAM file blocks can be received without the specification of locking if the purpose of input is reference. If blocks are received without the specification of locking, the input blocks may not be up-to-date because other UAPs may update the blocks during the input processing.

(4) Access to DAM files outside the transaction processing range (recoverable DAM files)

Processes outside the transaction range (before the transaction starts or after the acquisition of the synchronization point) can only receive blocks by accessing them for a reference purpose. Blocks cannot be locked.

(5) Transaction range for access to DAM files (recoverable DAM files)

When DAM files are accessed, they are locked and unlocked for each transaction branch or for each global transaction. For details on locking of DAM files, see *4.1.7 Locking DAM files*.

(6) Notes on access to DAM files exceeding 2 gigabytes

You cannot use DAM input or output functions (`dc_dam_get`, `dc_dam_put`, `dc_dam_read`, `dc_dam_rewrite`, `dc_dam_write`, `CBLDCDMB('GET')`, `CBLDCDMB('PUT')`, `CBLDCDMB('READ')`, `CBLDCDMB('REWT')`, and `CBLDCDMB('WRIT')`) to input or output DAM file data exceeding 2 gigabytes at one time. If you attempt to use one of these functions to input or output DAM file data exceeding 2 gigabytes, the function will return with a `DCDAMER_BUFFER` or 01604 error.

4.1.4 Access to a DAM file in online mode (operation from an SUP, SPP, or MHP)

To access a DAM file from a UAP in online mode (e.g., file reference or update), processing must be done in the transaction. If the DAM file open function is called before the transaction is started, terminate all the transactions started after the DAM file was opened, then close the DAM file.

(1) Name used when a DAM file is accessed

To open a DAM file, use the function `dc_dam_open()` [`CBLDCDAM ('OPEN')`] in which the logical file name is specified. When the DAM file is opened, the file descriptor is returned as the name for identifying the file. For processing after the file is opened, specify this file descriptor in the function to access the file (e.g., file input, update, or output). To close the DAM file, use the function `dc_dam_close()` [`CBLDCDAM('CLOS')`] in which the file descriptor is specified. Shut down the file

descriptor in the UAP even for processing after the file is opened.

A file descriptor becomes valid in the following cases:

- When the file is opened within the transaction processing range

  The file descriptor remains valid until one of the following events occurs:

  - Logical file is closed
  - Transaction synchronization point is acquired
  - UAP process is terminated

- When the file is opened outside the transaction processing range

  The file descriptor remains valid until one of the following events occurs:

  - Logical file is closed
  - UAP process is terminated

Use the logical file name to logically shut down the DAM file, release the DAM file from the shutdown state, or reference the status of the DAM file during processing.

(2) Inputting/outputting multiple blocks collectively

Consecutive blocks can be input/output collectively. When inputting/outputting a DAM file, specify an access block range as a structure in the corresponding function. The block range must be specified with the relative block number. More than one structure can be specified.

(3) Procedure for referencing/updating blocks

To reference a DAM file block, enter the block by using the function `dc_dam_read()` [`CBLDCDAM('READ')`]. At this time, you can also specify whether to allow another transaction to reference/update the block.

To update a DAM file block, enter the block by using the function `dc_dam_read()`, then call the function `dc_dam_rewrite()` [`CBLDCDAM ('REWT')`] to update the block.

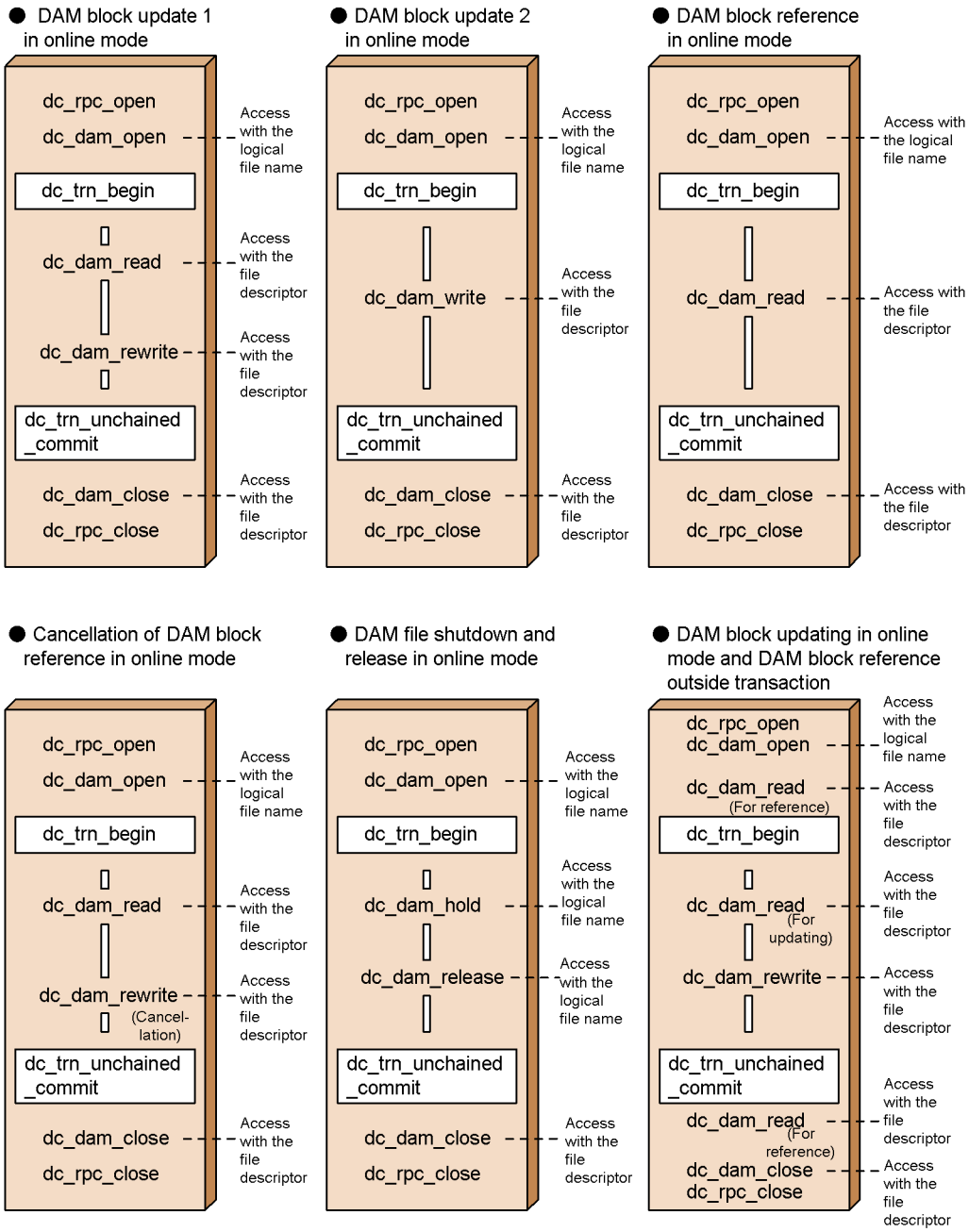
Call the function `dc_dam_write()` [`CBLDCDAM('WRIT')`] when you want to overwrite a block without entering a block from a DAM file.

(4) Logical shutdown and release of a DAM file

If a logical conflict is found during processing for a DAM file block, the UAP can call the function `dc_dam_hold()` [`CBLDCDAM('HOLD')`] to shut down the DAM file so that another UAP cannot access the DAM file. The UAP can also call the function `dc_dam_release()` [`CBLDCDAM ('RLES')`] to release the DAM file from the shutdown state.

The figure below shows the procedure for accessing DAM files in online mode.

Figure 4-2: Access to DAM files in online mode



(5) Access from transaction process to DAM file block

If an error occurs during access from a transaction process to a DAM file, call the function `abort()` from the UAP in order to terminate the transaction process abnormally.

Depending on the function that accessed the file previously, an access to a block may cause return with an error. The result of access (normal return or error return) varies depending on whether the function for the access is called within the same transaction or from a different transaction. Tables 4-1 and 4-2 list functions which can access DAM file blocks if particular functions were previously called.

Table 4-1: Functions able to access the same block in one transaction (recoverable DAM files)

Previously called function	Function to be called	Results or error return value
No function for accessing a DAM file has been called in the transaction	`dc_dam_read` (input for reference)	Y
	`dc_dam_read` (input for reference, lock specified)	Y
	`dc_dam_read` (input for update)	Y
	`dc_dam_rewrite` (update)	`DCDAMER_SEQER(01605)`
	`dc_dam_rewrite` (update cancellation)	`DCDAMER_SEQER(01605)`
	`dc_dam_write` (output)	Y
`dc_dam_read` (input for reference)	`dc_dam_read` (input for reference)	Y
	`dc_dam_read` (input for reference, lock specified)	Y
	`dc_dam_read` (input for update)	Y
	`dc_dam_rewrite` (update)	`DCDAMER_SEQER(01605)`
	`dc_dam_rewrite` (update cancellation)	`DCDAMER_SEQER(01605)`
	`dc_dam_write` (output)	Y

Previously called function	Function to be called	Results or error return value
dc_dam_read (input for reference, lock specified)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y
dc_dam_read (input for update)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	Y
	dc_dam_rewrite (update cancellation)	Y
	dc_dam_write (output)	DCDAMER_SEQER (01605)
dc_dam_rewrite (update)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y

Previously called function	Function to be called	Results or error return value
dc_dam_rewrite (update cancellation)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y
dc_dam_write (output)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y

Legend:

Y: No error

Table 4-2: Functions able to access the same block in different transaction (recoverable DAM files)

Previously called function	Function to be called	Results or error return value
No function for accessing a DAM file has been called in the transaction	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y
dc_dam_read (input for reference)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y
dc_dam_read (input for reference, lock specified)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	DCDAMER_EXCER (01602)#
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	DCDAMER_EXCER(01602)#

Previously called function	Function to be called	Results or error return value
dc_dam_read (input for update)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	DCDAMER_EXCER (01602)#
	dc_dam_read (input for update)	DCDAMER_EXCER (01602)#
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	DCDAMER_EXCER(01602)#
dc_dam_rewrite (update)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	DCDAMER_EXCER (01602)#
	dc_dam_read (input for update)	DCDAMER_EXCER (01602)#
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	DCDAMER_EXCER (01602)#
dc_dam_rewrite (update cancellation)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y

Previously called function	Function to be called	Results or error return value
dc_dam_write (output)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference lock specified)	DCDAMER_EXCER (01602)#
	dc_dam_read (input for update)	DCDAMER_EXCER (01602)#
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	DCDAMER_EXCER (01602)#

Legend:

Y: No error

#

Waits until released from the lock state if DCDAM_WAIT is set in flags.

(6) Reasons why DAM service functions return with a DCDAMER_PROTO error (recoverable DAM files)

DAM service functions return with a DCDAMER_PROTO error ("01600" in the case of COBOL) for the following reasons. The reason varies with the function called:

1. The function dc_rpc_open() (CBLDCRPC('OPEN ') in the case of COBOL) has not been called.
2. For recoverable DAM files, atomic_update=N is specified in the user service definition.
3. The UAP is not correctly linked as follows:
   - The library (-ltdam) to be used for access to a TAM file through the DAM service API is incorrectly linked.
   - The transaction control object file is incorrectly registered with the resource manager.
4. The function dc_dam_start() (CBLDCDAM('STRT') in the case of COBOL) has not been called when the atomic_update = N in the user service definition (for unrecoverable DAM files).

(7) Referencing the status of a DAM file

The status of a DAM file in use can be referenced in online mode. Call the function dc_dam_status() [CBLDCDAM('STAT')] to reference the status of a DAM file. This

function allows referencing of the following information:

- Number of logical file blocks
- Length of a logical file block
- Name of the physical file associated with a logical file
- Current status of a logical file (whether the file is shut down or not)
- Logical file attributes specified in the DAM service definition
- Logical file security attributes specified in the DAM service definition

(a) Note on using the function dc_dam_status()

When the function `dc_dam_status()` is called, the DAM service locks the file to acquire the information. Therefore, frequent use of this function may cause lock waits and reduced throughput. The status of a DAM file should be referenced in online mode as little as possible.

4.1.5 Access to a DAM file in offline mode (operation from a UAP that handles offline work)

The contents of a DAM file can be output under the batch environment. Close a physical file opened in offline mode as soon as processing terminates. After a physical file is opened, a mixture of input and output operations is not permitted. Before starting block input or output, close the physical file once. Functions for use offline cannot be used online (SUP, SPP, MHP). If you use these functions online, operation is not guaranteed.

(1) Name used when a DAM file is accessed

To open a physical file, use the function `dc_dam_iopen()` [`CBLDCDMB ('OPEN')`] in which the physical file name set upon the file allocation is specified. When the physical file is opened, the file descriptor is returned. Shut down the file descriptor in the UAP because the file descriptor is used for processing between when the file is opened and when the file is closed.

For processing after the file is opened, specify this file descriptor in the function to access the file. The processing includes block input (`dc_dam_get()`) and block output (`dc_dam_put()`). To close the physical file, use the function `dc_dam_iclose()` [`CBLDCDMB('CLOSE')`] in which the file descriptor is specified.

(2) Inputting/outputting blocks

When a physical file is opened using the function `dc_dam_iopen()`, its blocks can be input/output in one of the following two methods:

- Inputting/outputting blocks one after another from the head of a file

  Call the function `dc_dam_get()` [`CBLDCDMB('GET ')`] to input a block. Call the

function dc_dam_put() [CBLDCDMB('PUT ')] to output a block. After a physical file is opened, a combination of input and output operations is not permitted for UAP processing. Before starting block input or output, close the physical file once.

- Inputting/outputting arbitrary blocks

  To input/output an arbitrary block, specify OVERWRITE in the function dc_dam_iopen() before a physical file is opened. When the file is opened using another function, arbitrary blocks cannot be input/output.

  Arbitrary blocks can be input/output in one of the following two methods. Combining these methods is not permitted. Use either of these methods to input/output blocks:

  1. Seek the relative block number of a block to be input/output by calling the function dc_dam_bseek() [CBLDCDMB('BSEK')].

     After the appropriate number is found, use the function dc_dam_get() to input the block or the function dc_dam_put() to output the block. In this case, a combination of input and output operations is permitted for processing after the function dc_dam_iopen() is called and before the function dc_dam_iclose() is called.

  2. Specify the relative block number of a block to be input/output and call either of the following functions. A combination of input and output operations is permitted:

     - Block input:

       dc_dam_dget() [CBLDCDMB('DGET')]

     - Block output:

       dc_dam_dput() [CBLDCDMB('DPUT')]

Only the former method allows specifying that multiple blocks should be input/output collectively.

(3) Inputting/outputting multiple blocks collectively

When a physical file is allocated and opened, the number of blocks can be specified as an input/output unit.

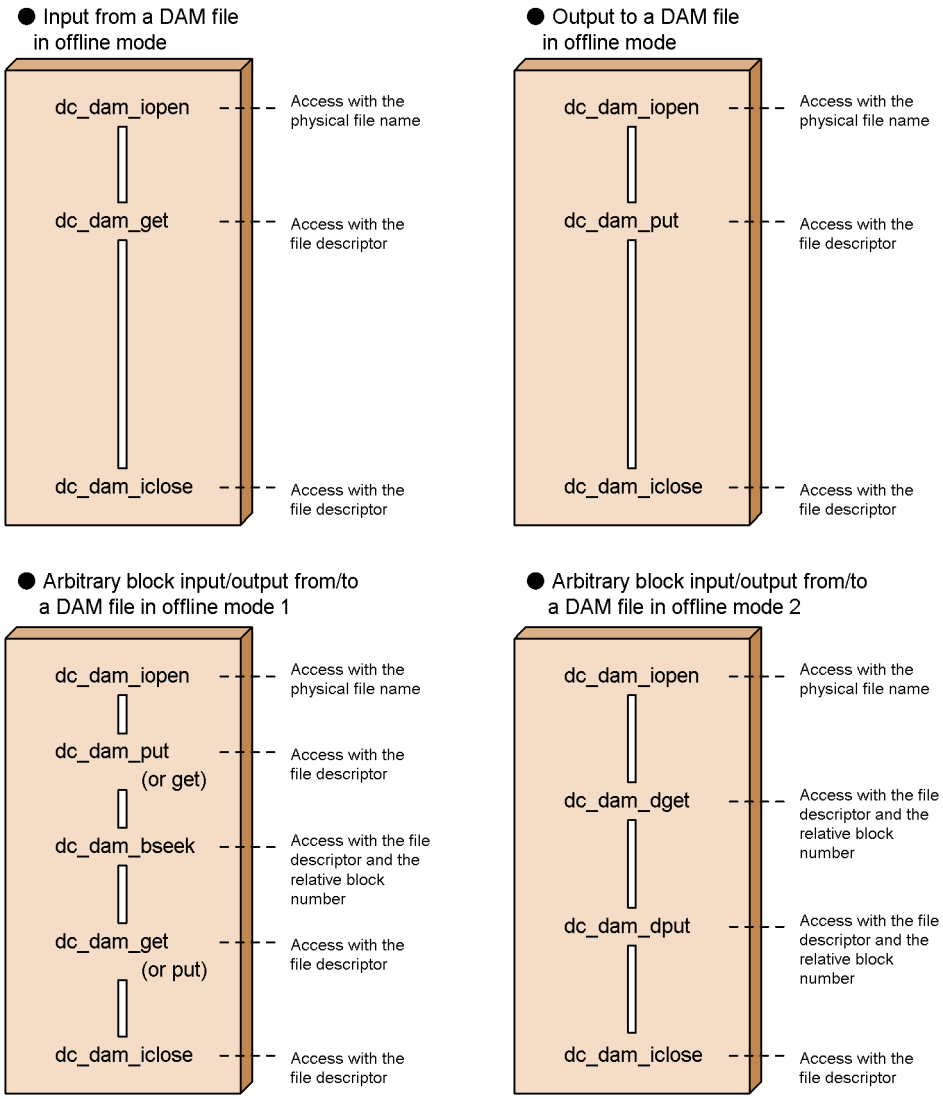
(4) Initializing/recreating blocks

When an input block is output to a physical file, it is possible to specify whether the area following the block to be output is to be padded with null characters. Specify INITIALIZE to pad the area following the block to be output with null characters. Otherwise, specify OVERWRITE. When OVERWRITE is specified, an arbitrary block can be input/output. In this case, the area following the output block is not updated and remains as it is.

Whether to fill the remaining blocks with null characters is specified in the argument to the function `dc_dam_iopen()`. This specification will be in effect when a block is output to the file using the function `dc_dam_put()` and the file is closed using the function `dc_dam_iclose()`. If the UAP is terminated without using the function `dc_dam_iclose()`, the remaining blocks are not filled with null characters. To fill the remaining blocks with null characters, be sure to call the function `dc_dam_iclose()`.

The figure below shows the procedure for accessing DAM files in offline mode.

Figure 4-3: Procedure for accessing DAM files in offline mode



4.1.6 Creating physical files (operation from a UAP that handles offline work)

Physical files are created in offline mode. Allocate a physical file to the OpenTP1 file system by using the function `dc_dam_create()` [`CBLDCDMB('CRAT')`]. Set the following information upon the allocation:

- Name of the physical file to be allocated

- Length of a block, and the number of blocks
- Number of blocks to be processed collectively, which is used as an input/output unit
- File owner, the owner group, and access right from another UAP

When a physical file is allocated, the file descriptor is returned.

The file descriptor is used for processing after the file is opened. The file descriptor returned by the function `dc_dam_create()` is available with the following functions:

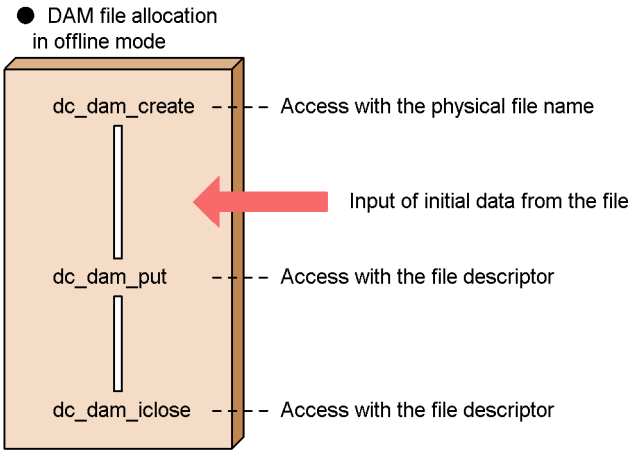
- `dc_dam_put()` (Output a block)
- `dc_dam_iclose()` (Close a file)

The file descriptor returned by the function `dc_dam_create()` is unavailable with the following functions and an error is returned if used:

- `dc_dam_get()` (Input a block)
- `dc_dam_dget()`, `dc_dam_dput()` (Input/Output directly a block)
- `dc_dam_bseek()` (Seek a block)

The figure below shows the procedure for creating a DAM file as the first physical file in the OpenTP1 file system.

Figure 4-4: Procedure for creating DAM file



4.1.7 Locking DAM files

Suppose that, when a DAM file is being updated, an interrupt for updating the same file comes from another UAP. Two updates would then be reflected on the same logical file and would cause inconsistencies in the file. To avoid this, the function to access a file can contain a specification for locking the file. Through this lock control, data

consistency can be assured on DAM files even when they are accessed from more than one UAP.

- Transaction range for access to DAM files (recoverable DAM files)

  When DAM files are accessed, they can be locked for each transaction branch. This could cause the following situation: Access from multiple transaction branches belonging to one global transaction to the same block or file could cause a lock error. To avoid this, it is also possible to lock files for each global transaction. To use this type of locking, the DAM service definition for the DAM files must include the specification that the files be accessed from each global transaction.

  When locking for each global transaction is in effect, access from a transaction branch to a DAM file will not be parallel, but sequential. This could lower the transaction performance. If DAM files are to be accessed in parallel from each transaction branch, locking for each global transaction must not be specified.

(1) Lock modes

Lock conditions for accessing DAM files are called *lock modes*. The following lock modes are available:

Lock for reference (shared mode PR Protected Retrieve):

The UAP can only reference files with lock specified. Other transactions are permitted only to reference the files.

Lock for update (exclusive mode EX EXclusive):

The UAP can reference and update files with lock specified. Other transactions are not permitted to reference or update the files.

(2) Lock units (recoverable DAM files)

Lock can be specified in units of blocks or files when a DAM file is accessed in online mode as explained below.

(a) Block-based locks

Lock is enabled in blocks. When a block is referenced, the lock of the shared mode is enabled. When a block is updated or output, the lock of the exclusive mode is enabled. The specification of lock for reference can be disabled by specifying no lock in the option (other UAPs allowed to reference/update blocks). The specification of the acquired lock is reset when the transaction processing that specified processing for the DAM file terminates normally.

(b) File-based lock

Each logical file can be locked. If locking of a logical file is specified, the entire file will be locked during the period from the time the file is opened to the time the transaction process terminates normally.

File-based lock can be specified in the following condition:

- Locking for each transaction branch is specified and the logical file was opened within the transaction range.

File-based lock cannot be specified in the following cases. Use block-based lock.

- The logical file was opened outside the transaction range.
- Locking for each global transaction was specified.

(3) Specification of waiting for a resource to be released from lock

- `dc_dam_open()`

  If an attempt is made to input data from or output data to a block which is locked by another transaction (lock error), the function for this access will return with an error or wait until the block is unlocked. This can be specified in the argument to the function `dc_dam_open()` for opening the DAM file.

  If a DAM file is opened under file-based locks, waiting for release from lock (lock wait type) cannot be specified.

  If a lock error occurs when a file is opened by using the function `dc_dam_open()`, the function unconditionally returns with the error.

- `dc_dam_read()` and `dc_dam_write()`

  The functions `dc_dam_read()` and `dc_dam_write()` can specify whether, when a lock error occurs, it will return or wait until the resource is unlocked. If this specification is omitted, the value specified in the function `dc_dam_open()` is assumed.

  If wait until unlocking is specified and a deadlock or timeout occurs, deadlock information will be output after the function waiting for an unlocked resource returns with an error. If the function returns with a deadlock or timeout error, acquire the synchronization point of the transaction and free all the acquired resources.

(4) Lock in online mode and offline mode

A DAM file being used in online mode cannot be accessed in offline mode. To access a DAM file, being used in online mode, in offline mode, use the `damhold` and `damrm` command to switch the online mode to the offline mode. Then, use the `damadd` command to switch the DAM file back into the online mode.

Even in offline mode, different UAPs cannot access one DAM file at the same time. The UAP process that has opened a DAM file uses it exclusively until it is closed.

4.1.8 Access to unrecoverable DAM files

You can create a DAM file which does not guarantee consistency management or error

recovery through transactions. This DAM file is called an unrecoverable DAM file. Unrecoverable DAM file blocks can be updated using the functions `dc_dam_write()` and `dc_dam_rewrite()` instead of transaction processing.

(1) Definition of unrecoverable DAM files

To define an unrecoverable DAM file, specify the definition command `damfile` with the `-n` option in the DAM service definition.

(2) Access to unrecoverable DAM files

Before accessing a file, call the function `dc_dam_start()` [`CBLDCDAM ('STRT')`]. When completing the file access, call the function `dc_dam_end()` [`CBLDCDAM('END ')`]. When the function `dc_dam_start()` is called, the function `dc_dam_end()` must be called after the file access is terminated.

An unrecoverable DAM file is accessed using a DAM service function. An unrecoverable DAM file can be accessed just like a recoverable DAM file.

The file descriptor returned when a file is opened remains valid until one of the following events occurs:

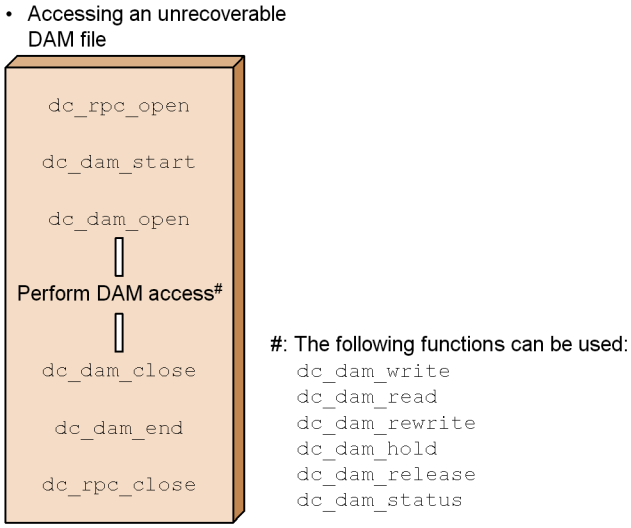
- Logical file is closed
- Use of an unrecoverable DAM file (a call to the function `dc_dam_end()`) is terminated
- UAP process is terminated

(3) Action to be taken in case of file access error

Even if an error occurs during the access to an unrecoverable DAM file, the file data error cannot be recovered.

The figure below shows the procedure for accessing an unrecoverable DAM file.

Figure 4-5: Procedure for accessing unrecoverable DAM file



(4) Relationship between functions for accessing unrecoverable DAM files

Depending on the function that accessed the file previously, an access to a block even from the same UAP process may cause return with an error. The result of access (normal return or error return) varies depending on whether the function for the access is called from the same UAP or from a different UAP. Tables 4-3 and 4-4 list functions which can access DAM file blocks if particular functions were previously called.

Table 4-3: Functions able to access the same block in one UAP (unrecoverable DAM files)

Previously called function	Function to be called	Results or error return value
No function for accessing a DAM file has been called after the function `dc_dam_start()` was called	`c_dam_read` (input for reference)	Y
	`dc_dam_read` (input for reference, lock specified)	Y
	`c_dam_read` (input for update)	Y
	`c_dam_rewrite` (update)	`DCDAMER_SEQER` (`01605`)
	`dc_dam_rewrite` (update cancellation)	`DCDAMER_SEQER` (`01605`)
	`dc_dam_write` (output)	Y

Previously called function	Function to be called	Results or error return value
dc_dam_read (input for reference)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y
dc_dam_read (input for reference, lock specified)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y
dc_dam_read (input for update)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	Y
	dc_dam_rewrite (update cancellation)	Y
	dc_dam_write (output)	Y
dc_dam_rewrite (update)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y

Previously called function	Function to be called	Results or error return value
dc_dam_rewrite (update cancellation)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y
dc_dam_write (output)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y

Legend:

Y: No error

Table 4-4: Functions able to access the same block in different UAP (unrecoverable DAM files)

Previously called function	Function to be called	Results or error return value
No function for accessing a DAM file has been called after the function dc_dam_start() was called	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y

Previously called function	Function to be called	Results or error return value
dc_dam_read (input for reference)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y
dc_dam_read (input for reference, lock specified)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y
dc_dam_read (input for update)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	DCDAMER_EXCER (01602)#
	dc_dam_read (input for update)	DCDAMER_EXCER (01602)#
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	DCDAMER_EXCER (01602)#
dc_dam_rewrite (update)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y

Previously called function	Function to be called	Results or error return value
dc_dam_rewrite (update cancellation)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference, lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y
dc_dam_write (output)	dc_dam_read (input for reference)	Y
	dc_dam_read (input for reference lock specified)	Y
	dc_dam_read (input for update)	Y
	dc_dam_rewrite (update)	DCDAMER_SEQER (01605)
	dc_dam_rewrite (update cancellation)	DCDAMER_SEQER (01605)
	dc_dam_write (output)	Y

Legend:

Y: No error

#

Waits until released from the lock state if DCDAM_WAIT is set in flags.

(5) Locking unrecoverable DAM files

As in the case of recoverable DAM files, unrecoverable DAM files can also be locked. The following gives an explanation of locking unrecoverable DAM files. For comparison between recoverable and unrecoverable DAM files, see Item (6) in *4.1.8 Access to unrecoverable DAM files*.

(a) Unrecoverable DAM file locking range

Unrecoverable DAM files can be accessed regardless of the transaction processing range.

(b) Lock modes

The lock modes for unrecoverable DAM files are the same as for recoverable DAM files. For details, see (1) in *4.1.7 Locking DAM files*.

(c) Lock units

The lock units for unrecoverable DAM files are the same as for recoverable DAM files. For details, see (2) in *4.1.7 Locking DAM files*.

(d) Specification of waiting for a resource to be released from lock

The specification of waiting for a resource to be released from lock in the case of unrecoverable DAM files is the same as in the case of recoverable DAM files except for the following point:

- The lock wait type specified in the function `dc_dam_open()` includes the specification of handling this function itself. If a lock error occurs when a recoverable DAM file is opened by using the function `dc_dam_open()`, the function unconditionally returns with an error. In the case of an unrecoverable DAM file, the function can proceed according to the lock wait type specified in its argument.

  For details, see (3) in *4.1.7 Locking DAM files*.

(6) Comparison between recoverable and unrecoverable DAM files

The following explains the comparison between recoverable and unrecoverable DAM files. Table 4-5 lists the differences in file access. Table 4-6 lists the differences in the locking range upon file access.

Table 4-5: Differences in access to recoverable and unrecoverable DAM files

DAM service function	Conditions for calling function	DAM file types and access positions		
		Recoverable DAM file		Unrecoverable DAM file
		Outside transaction processing range	Within transaction processing range	
`dc_dam_open`	File-based lock, lock wait	N	Y	Y
	File-based lock, immediate return	N	Y	Y
	Block-based locks, lock wait	Y	Y	Y
	Block-based locks, immediate return	Y	Y	Y
`dc_dam_close`	Open a file within the transaction processing range	Y	Y	Y
	Open a file outside the transaction processing range	--	N	Y
`dc_dam_read`	Input for reference, no lock	Y	Y	Y

DAM service function	Conditions for calling function	DAM file types and access positions		
		Recoverable DAM file		Unrecoverable DAM file
		Outside transaction processing range	Within transaction processing range	
	Input for reference, lock specified	N	Y	Y
	Input for update, lock specified	N	Y	Y
dc_dam_rewrite	Output for update	N	Y	Y
	Update cancellation	N	Y	Y
dc_dam_write	No condition	N	Y	Y

Legend:

Y: Can be used for DAM files.

N: Returns with an error if called for DAM files.

--: Cannot be used for DAM files.

Table 4-6: Differences in locking range upon access to recoverable and unrecoverable DAM files

Lock unit# and function to be called		Lock mode	Recoverable DAM file	Unrecoverable DAM file
File-based lock	dc_dam_open	EX	• Locked until termination of synchronization point processing	• Locked until termination of processing through dc_dam_close() or dc_dam_end()

Lock unit# and function to be called		Lock mode	Recoverable DAM file	Unrecoverable DAM file
Block-based locks	`dc_dam_read` (reference)	PR	• Locked until termination of synchronization point processing	• Locked until termination of processing through `dc_dam_read()`
	`dc_dam_read` (update)	EX	• Locked until termination of synchronization point processing • Locked until termination of processing through `dc_dam_rewrite()` (cancellation)	• Locked until termination of processing through `dc_dam_rewrite()` (update or cancellation)
	`dc_dam_write`	EX	• Locked until termination of synchronization point processing	• Locked until termination of processing through `dc_dam_write()`

#

This lock unit means that specified in the function `dc_dam_open()`. When file-based lock is specified in this function, the lock units specified in the functions `dc_dam_read()` and `dc_dam_write()` are invalid.

4.1.9 Interchangeability of DAM and TAM services

DAM file service functions can be used to access TAM file records. See *4.2.9 Interchangeability of TAM and DAM services* for details.

4.2 TAM file service (TP1/FS/Table Access)

This section explains TAM files which can be used as user files dedicated to OpenTP1. The use of TAM files allows high-speed access to direct files.

Before TAM files can be used, TP1/FS/Table Access must be installed in the system and the basic OpenTP1 facility must be TP1/Server Base. The TAM file service is unavailable with TP1/LiNK.

4.2.1 TAM file configuration

A TAM file comprises data items. Each data item is called a *record*. The TAM file keys and records are loaded into the memory. Fast access to a file from a UAP is enabled by accessing the key in the memory instead of accessing the actual file. The index part and data part are called a *TAM table*.

A TAM table consists of the following two parts:

- Index part which contains the keys corresponding to records
- Data part which contains records

(1) Index types of TAM tables

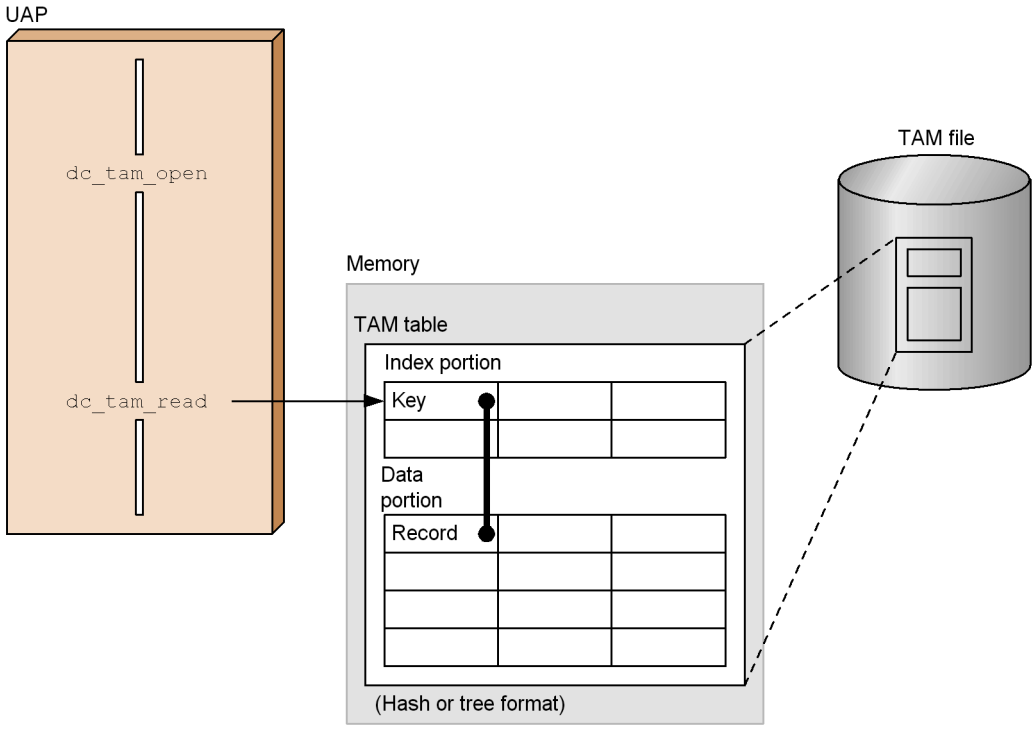
Index types are categorized into a hash format and a tree format. A method of corresponding keys to records is different between the hash format and the tree format. When accessing a TAM table, verify the index type of the TAM table, then create a UAP. If access is made to a TAM table of a different index type, the UAP TAM access function returns with an error.

(2) Environment for access to TAM tables

TAM tables can be accessed only in online environment. TAM tables cannot be accessed in offline environment. To access a TAM table from a UAP, use the access mode specified in the TAM service definition. If another access mode is used, the UAP TAM access function returns with an error.

The figure below shows the TAM file configuration.

Figure 4-6: TAM file configuration



4.2.2 Conditions for accessing a TAM table

Only the TAM table of a TAM file which exists at the node (machine) of the corresponding transaction branch can be accessed. Processing for TAM tables of each node is done independently of each other. Thus, TAM table names are managed for each node. When accessing a TAM table in a global transaction, use the table name in the node.

(1) Access to a TAM table from a UAP and transaction functions

The TAM table open and close functions can be used regardless of whether a transaction has been started. However, functions other than the TAM file open and close functions (e.g., table reference and update functions) must be used after a transaction is started. If a TAM file was opened before a transaction was started, terminate all the transactions started after the TAM file was opened, then close the TAM file.

(2) Access to a TAM file and RPC modes

To access a TAM file, all the RPC modes of the global transactions must be of synchronous response type. Operation is not ensured if a TAM table is accessed from

an asynchronous-response-type RPC or nonresponse-type RPC.

4.2.3 Name used when a TAM table is accessed

A TAM table is opened with the TAM table name. When a TAM table is opened, the table descriptor is returned as the name for identifying the table. For processing after the TAM table is opened, specify the table descriptor in the function to access the table. The processing includes record input, update, addition, and deletion.

4.2.4 Procedure for accessing a TAM table

(1) Opening TAM tables

When a UAP in C language is used, use the function `dc_tam_open()` to open a TAM table.# Call the function `dc_tam_open()` for each UAP.

A TAM table can be opened both inside and outside the transaction range. However, if a TAM table is opened before a transaction is started, lock cannot be specified for the table. See *4.2.6 Lock for TAM tables* for details on lock for TAM files.

To close a TAM table, specify the table descriptor in the corresponding function.# Keep the table descriptor in the UAP because the table descriptor is used for processing after the table is opened.

#

When the COBOL language is used for UAP coding, there is no need to open and close the TAM table. The TAM table is opened when it is accessed. The TAM table is closed when the transaction is completed.

(2) Procedures for record input/update/addition/deletion

To input a TAM table record for reference or update processing, call the function `dc_tam_read()` [`CBLDCTAM('FxxR')('FxxU')('VxxR')('VxxU')`]. At this time, whether to permit reference or update processing from another global transaction can be specified.

To update a TAM table record, input the record with the function `dc_tam_read()`, then call the function `dc_tam_rewrite()` [`CBLDCTAM('MFY ')('MFYS')('STR ')('WFY ')('WFYS')('YTR ')`]. (Update on the assumption of input)

To overwrite an existing record or add a new record instead of inputting a record from the TAM table, call the function `dc_tam_write()` [`CBLDCTAM('MFY ')('MFYS')('STR ')('WFY ')('WFYS')('YTR ')`].

To delete a record from the TAM table, call the function `dc_tam_delete()` [`CBLDCTAM('ERS ')('ERSR')('BRS ')('BRSR')`]. The record to be deleted can be saved in the buffer of any address. Specify the save destination address in the function `dc_tam_delete()`.

If beginnings of the buffer area for the function `dc_tam_read()` or

`dc_tam_delete()` and the data area for the function `dc_tam_rewrite()` or `dc_tam_write()` are specified on 4-byte boundaries, higher-speed access can be achieved than when such specifications are not given.

(3) Inputting/outputting multiple records collectively

Multiple key values (records) can be input/output collectively. When inputting/outputting a TAM table, specify an access key value as a structure in the corresponding function. More than one structure can be specified.

(4) Record input according to index types

When a record is input from a TAM table, the retrieval type to be specified depends on the index type.

■ With the hash format

First retrieval and NEXT retrieval are available. These retrieval methods enable you to retrieve all records. Call the first `dc_tam_read()` (with the first record specified), then input the first record. The records following the key value are input in the NEXT retrieval sequence specified in the `dc_tam_read()`.

You can use the all-record retrieval method to delete all records from a TAM table. To delete all records from a TAM table:

1. Use first retrieval to acquire the first record.
2. Use NEXT retrieval with the first record specified as the key, to acquire the next record.
3. Delete the first record.
4. Use NEXT retrieval with the currently acquired record specified as the key, to acquire the next record.
5. Delete the record that you specified as the key in step 4.
6. Repeat steps 4 and 5 until there is no next record.
7. When there is no next record, delete the record that you specified as the key in the last NEXT retrieval.

This method requires you to specify a key indicating the start position for retrieval. Thus the system does not search the empty hash area extending from the start to the position immediately preceding the specified key. This makes for an efficient retrieval process.

The following method uses a large proportion of the CPU's capacity, so the response may be delayed.

1. Use first retrieval to acquire the first record.
2. Delete the first record.

3. Repeat steps 1 and 2 until there are no more records.

First retrieval searches the hash area from the start. Each time you execute retrieval, this method searches the record from the start, including the hash area that became empty when you deleted the records in the previous processes. It is therefore an inefficient method and the response may be delayed.

- **With the tree format**

  Retrieval of =, <=, >=, <, and > is available for the specified key value. Input the record corresponding to the key value. To input the records of multiple keys in a range, specify =, <=, >=, <, and > so that records satisfying the conditions can be input subsequently.

(5) Closing TAM tables

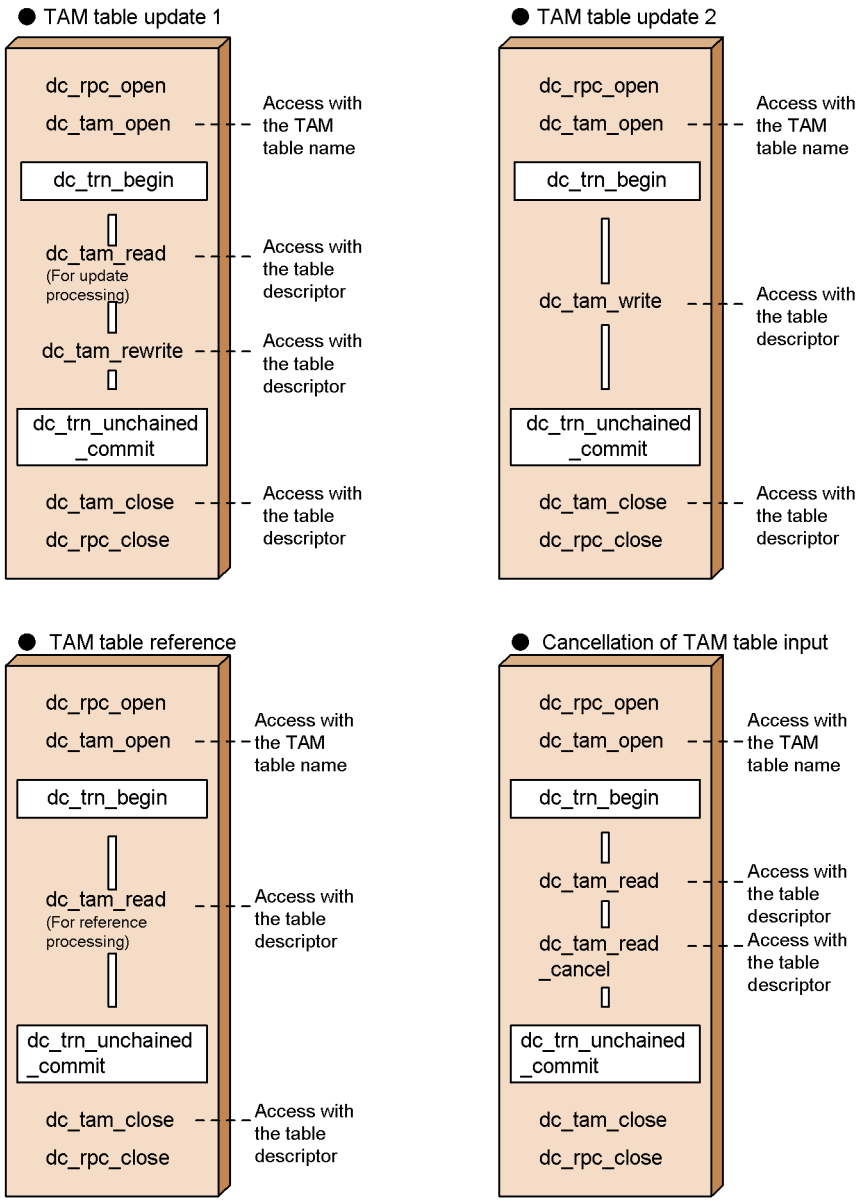
Use the function `dc_tam_close()` to close TAM tables.

If a TAM table was opened in the transaction range, close the TAM table in the transaction. If the transaction was terminated without the close function called, OpenTP1 closes the TAM table.

If a TAM table was opened outside the transaction range, close the TAM table outside the transaction. If the function `dc_tam_close()` is called in the transaction, the function returns with an error.

The figure below shows the procedures for accessing TAM tables.

Figure 4-7: Access to TAM tables



(6) Acquiring TAM table status

Use the function `dc_tam_get_inf()` [`CBLDCTAM('GST ')`] to acquire the status of a TAM table in online mode. The function `dc_tam_get_inf()` can be called both

inside and outside the transaction range. The function `dc_tam_get_inf()` returns the following statuses of the TAM table:

- Open status
- Close status
- Logical shutdown status
- Shutdown status due to an error

The function `dc_tam_get_inf()` returns the open status when the UAP that called the function `dc_tam_get_inf()` has not opened the TAM table, but another UAP has opened the specified TAM table.

(7) Acquiring TAM table information

Use the function `dc_tam_status()` [`CBLDCTAM('INFO')`] to acquire the information of a TAM table in online mode. The function `dc_tam_status()` can be called both inside and outside the transaction range. The function `dc_tam_status()` returns the following information of the TAM table:

- TAM file name
- TAM table status
- Number of records in use
- Maximum number of records
- Index type
- Access type
- Loading opportunity
- TAM record length
- Key length
- Key start position
- Security attribute

4.2.5 Relationship between transactions and TAM access

If a TAM access error occurs in a transaction branch, call the `abort()` from the UAP in order to terminate the global transaction process abnormally.

Even if access is made to records in the same global transaction, an error might be returned upon access to a record. This is caused by the function called for the previous access. Also, even when access is made to the same record, the access results are different between when the record belongs to the same global transaction and when the record belongs to a different global transaction. Tables 4-7 and 4-8 show the

processing results when a function was called more than once for the same record.

Table 4-7: Processing results when function was called more than once for the same record (in one global transaction)

Previously called function	Function to be called	Results or error return value
No function for accessing a TAM table has been called in the transaction	`dc_tam_read` (input for reference)	Y
	`dc_tam_read` (input for reference, lock specified)	Y
	`dc_tam_read` (input for update)	Y
	`dc_tam_read_cancel` (input cancellation)	`DCTAMER_SEQENCE` (`01732`)
	`dc_tam_rewrite` (update on the assumption of input)	`DCTAMER_SEQENCE` (`01732`)
	`dc_tam_write` (update)	Y
	`dc_tam_write` (addition)	Y
	`dc_tam_delete` (deletion)	Y
`dc_tam_read` (input for reference)	`dc_tam_read` (input for reference)	Y
	`dc_tam_read` (input for reference, lock specified)	Y
	`dc_tam_read` (input for update)	Y
	`dc_tam_read_cancel` (input cancellation)	`DCTAMER_SEQENCE` (`01732`)
	`dc_tam_rewrite` (update on the assumption of input)	`DCTAMER_SEQENCE` (`01732`)
	`dc_tam_write` (update)	Y
	`dc_tam_write` (addition)	`DCTAMER_EXKEY` (`01735`)
	`dc_tam_delete` (deletion)	Y

Previously called function	Function to be called	Results or error return value
dc_tam_read (input for reference, lock specified)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	Y
	dc_tam_read (input for update)	Y
	dc_tam_read_cancel (input cancellation)	Y#1
	dc_tam_rewrite (update on the assumption of input)	DCTAMER_SEQENCE (01732)
	dc_tam_write (update)	Y
	dc_tam_write (addition)	DCTAMER_EXKEY (01735)
	dc_tam_delete (deletion)	Y
dc_tam_read (input for update)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	Y
	dc_tam_read (input for update)	Y
	dc_tam_read_cancel (input cancellation)	Y
	dc_tam_rewrite (update on the assumption of input)	Y
	dc_tam_write (update)	Y
	dc_tam_write (addition)	DCTAMER_EXKEY (01735)
	dc_tam_delete (deletion)	Y

Previously called function	Function to be called	Results or error return value
dc_tam_read_ cancel (input cancellation)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	Y
	dc_tam_read (input for update)	Y
	dc_tam_read_cancel (input cancellation)	DCTAMER_SEQENCE (01732)[#2]
	dc_tam_rewrite (update on the assumption of input)	DCTAMER_SEQENCE (01732)
	dc_tam_write (update)	Y
	dc_tam_write (addition)	DCTAMER_EXKEY (01735)
	dc_tam_delete (deletion)	Y
dc_tam_rewrite (update on the assumption of input)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	Y
	dc_tam_read (input for update)	Y
	dc_tam_read_cancel (input cancellation)	DCTAMER_EXREWRT (01734)
	dc_tam_rewrite (update on the assumption of input)	Y
	dc_tam_write (update)	Y
	dc_tam_write (addition)	DCTAMER_EXKEY (01735)
	dc_tam_delete (deletion)	Y

Previously called function	Function to be called	Results or error return value
dc_tam_write (update or addition)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	Y
	dc_tam_read (input for update)	Y
	dc_tam_read_cancel (input cancellation)	DCTAMER_SEQENCE (01732)
	dc_tam_rewrite (update on the assumption of input)	DCTAMER_SEQENCE (01732)
	dc_tam_write (update)	Y
	dc_tam_write (addition)	DCTAMER_EXKEY (01735)
	dc_tam_delete (deletion)	Y
dc_tam_delete (deletion)	dc_tam_read (input for reference)	DCTAMER_NOREC (01731)
	dc_tam_read (input for reference, lock specified)	DCTAMER_NOREC (01731)
	dc_tam_read (input for update)	DCTAMER_NOREC (01731)
	dc_tam_read_cancel (input cancellation)	DCTAMER_NOREC (01731)
	dc_tam_rewrite (update on the assumption of input)	DCTAMER_NOREC (01731)[#3]
	dc_tam_write (update)	DCTAMER_NOREC (01731)
	dc_tam_write (addition)	Y
	dc_tam_delete (deletion)	DCTAMER_NOREC (01731)

Legend:

Y: No error

DCTAMER_NOREC (01731): The specified record is not found.

DCTAMER_SEQENCE (01732): The function dc_tam_read() has not been called.

DCTAMER_EXREWRT (01734): The table descriptor was updated by the dc_tam_rewrite().

DCTAMER_EXKEY (01735): Addition is not permitted because there is the record of the key value specified in the function.

#1

DCTAMER_EXREWRT or DCTAMER_EXWRITE is returned in the following case:

Before the function dc_tam_read() (input for reference, lock specified) is called, the function dc_tam_rewrite() or the function dc_tam_write() has been called to update or add a record.

#2

DCTAMER_EXWRITE is returned in the following case:

Before the function dc_tam_read_cancel() (input cancellation) is called, the function dc_tam_rewrite() or the function dc_tam_write() has been called to update or add a record.

#3

DCTAMER_SEQENCE is returned in the following case:

Before the function dc_tam_delete() (deletion) is called, the function dc_tam_write() has been called to add a record.

Table 4-8: Processing results when function was called more than once for the same record (in a different global transaction)

Previously called function	Function to be called	Results or error return value
No function for accessing a TAM table has been called in the transaction.	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	Y
	dc_tam_read (input for update)	Y
	dc_tam_read_cancel (input cancellation)	--#1
	dc_tam_rewrite (update on the assumption of input)	--#1
	dc_tam_write (update)	Y
	dc_tam_write (addition)	Y
	dc_tam_delete (deletion)	Y

Previously called function	Function to be called	Results or error return value
dc_tam_read (input for reference)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	Y[#2]
	dc_tam_read (input for update)	Y[#2]
	dc_tam_read_cancel (input cancellation)	--[#1]
	dc_tam_rewrite (update on the assumption of input)	--[#1]
	dc_tam_write (update)	Y[#2]
	dc_tam_write (addition)	DCTAMER_EXKEY (01735)
	dc_tam_delete (deletion)	Y[#2]
dc_tam_read (input for reference, lock specified)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	Y[#2]
	dc_tam_read (input for update)	DCTAMER_LOCK (01736)[#3]
	dc_tam_read_cancel (input cancellation)	--[#1]
	dc_tam_rewrite (update on the assumption of input)	--[#1]
	dc_tam_write (update)	DCTAMER_LOCK (01736)[#3]
	dc_tam_write (addition)	DCTAMER_LOCK (01736)[#3]
	dc_tam_delete (deletion)	DCTAMER_LOCK (01736)[#3]

Previously called function	Function to be called	Results or error return value
dc_tam_read (input for reference)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	DCTAMER_LOCK (01736)[#3]
	dc_tam_read (input for update)	DCTAMER_LOCK (01736)[#3]
	dc_tam_read_cancel (input cancellation)	--[#1]
	dc_tam_rewrite (update on the assumption of input)	--[#1]
	dc_tam_write (update)	DCTAMER_LOCK (01736)[#3]
	dc_tam_write (addition)	DCTAMER_LOCK (01736)[#3]
	dc_tam_delete (deletion)	DCTAMER_LOCK (01736)[#3]
dc_tam_read_cancel (input cancellation)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	Y[#4, #5]
	dc_tam_read (input for update)	Y[#4, #5]
	dc_tam_read_cancel (input cancellation)	--[#1]
	dc_tam_rewrite (update on the assumption of input)	--[#1]
	dc_tam_write (update)	Y[#4, #5]
	dc_tam_write (addition)	DCTAMER_LOCK (01736)[#3]
	dc_tam_delete (deletion)	DCTAMER_LOCK (01736)[#3]

Previously called function	Function to be called	Results or error return value
dc_tam_rewrite (update on the assumption of input)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	DCTAMER_LOCK (01736)[#3]
	dc_tam_read (input for update)	DCTAMER_LOCK (01736)[#3]
	dc_tam_read_cancel (input cancellation)	--[#1]
	dc_tam_rewrite (update on the assumption of input)	--[#1]
	dc_tam_write (update)	DCTAMER_LOCK (01736)[#3]
	dc_tam_write (addition)	DCTAMER_LOCK (01736)[#3]
	dc_tam_delete (deletion)	DCTAMER_LOCK (01736)[#3]
dc_tam_write (update, or addition)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	DCTAMER_LOCK(01736)[#3]
	dc_tam_read (input for update)	DCTAMER_LOCK (01736)[#3]
	dc_tam_read_cancel (input cancellation)	--[#1]
	dc_tam_rewrite (update on the assumption of input)	--[#1]
	dc_tam_write (update)	DCTAMER_LOCK (01736)[#3]
	dc_tam_write (addition)	DCTAMER_LOCK (01736)[#3]
	dc_tam_delete (deletion)	DCTAMER_LOCK (01736)[#3]

Previously called function	Function to be called	Results or error return value
dc_tam_delete (deletion)	dc_tam_read (input for reference)	Y
	dc_tam_read (input for reference, lock specified)	DCTAMER_LOCK (01736)#3
	dc_tam_read (input for update)	DCTAMER_LOCK (01736)#3
	dc_tam_read_cancel (input cancellation)	--#1
	dc_tam_rewrite (update on the assumption of input)	--#1
	dc_tam_write (update)	DCTAMER_LOCK (01736)#3
	dc_tam_write (addition)	DCTAMER_LOCK (01736)#3
	dc_tam_delete (deletion)	DCTAMER_LOCK (01736)#3

Legend:

Y: No error

--: Not applicable

DCTAMER_EXKEY (01735): Addition is not permitted because there is the record of the key value specified in the function.

DCTAMER_LOCK (01736): An lock error occurred.

#1: Another processing is executed in a different transaction.

#2: If another global transaction has added a record to or deleted a record from the same TAM table, DCTAMER_LOCK (01736) is returned. If DCTAM_WAIT is specified as the lock wait type, the wait-for release mode goes into effect.

#3: When DCTAM_WAIT is specified for the lock wait type, the function waits until the record is released from lock.

#4: DCTAMER_LOCK (01736) is returned if a record has been added to or deleted from the same TAM table in another transaction. However, when DCTAM_WAIT is specified for the lock wait type, the function waits until the record is released from lock.

#5: DCTAMER_LOCK (01736) is returned in the following case:

- Before the function dc_tam_read_cancel() is called, the function dc_tam_rewrite() or the function dc_tam_write() has been called to update or add a record in another global transaction. However, when DCTAM_WAIT is specified for the lock wait type, the function waits until the record is released from lock.

4.2.6 Lock for TAM tables

If another UAP starts TAM table update processing while a TAM file is being updated, the results of both update processing are concurrently reflected into one record. Consequently, the table contents include conflicting information. To prevent this problem, specify lock in the function to access the TAM file. Specifying lock ensures that the data items accessed from UAPs match each other.

Lock for DAM files is managed in global transactions.

(1) Lock modes

Lock conditions for accessing TAM tables are called lock modes. The following lock modes are available:

Lock for reference (shared mode PR Protected Retrieve):

Only records with lock specified can be referenced. Only reference from another global transaction is permitted.

Lock for update (exclusive mode EX EXclusive):

Records or tables with lock specified can be referenced and updated. Reference or update processing from another global transaction is not permitted.

(2) Lock units

Lock can be specified in units of records or tables when a TAM table is accessed in online mode as explained below.

(a) Record-based lock

Lock is enabled in records. When a record is input for reference processing, specify whether to enable lock for reference or not (other UAPs allowed to do update processing). When a record is input or updated for update processing, specify lock for update. The acquired lock is reset when the transaction that specified processing for the TAM table terminates normally.

(b) Table-based lock

Lock is enabled in tables. When a TAM table is opened under tables-based lock or when a record is added/deleted, specify lock for update for the entire TAM table. The acquired lock is reset when the transaction that specified processing for the TAM table terminates normally. If tables are opened before a transaction is started, tables-based lock cannot be specified for the tables.

(3) Specification for awaiting unlocking of resources

If an attempt is made to access a TAM table which is locked by another UAP (lock error), the function for this access will return with an error or wait until the TAM table is unlocked. This can be specified in the argument to the function.

If wait until unlocking is specified and a deadlock or timeout occurs, deadlock information will be output after the function waiting for an unlocked TAM table returns with an error. If the function returns with a deadlock or timeout error, acquire the synchronization point of the transaction and free all the acquired resources.

For UAPs written in COBOL, use either of the following methods to specify whether the function for access will wait until the TAM table is unlocked:

- Operand `tam_cbl_level` in the TAM service definition
- Setting of data-name-I for the CBLDCTAM

See the manuals *OpenTP1 System Definition* and *OpenTP1 Programming Reference COBOL Language* on how to specify for COBOL UAPs whether the function for access will wait until the resource is unlocked.

The table below indicates how lock specifications in TAM service functions are related to actual lock statuses. UAPs written in COBOL lock or unlock resources by using API functions for record access.

Table 4-9: Lock specifications in TAM service functions and actual lock statuses

Value assigned to flags on TAM service function			TAM table lock	TAM record lock
`dc_tam_open`	Table-based lock		Update lock	--[#1]
	Record-based lock		A record is locked using a function for access to the record.	
`dc_tam_read`	For referencing	No lock	--	--
		Lock possible	Lock for referencing[#2]	Lock for referencing
	For update		Lock for referencing[#2]	Update lock
`dc_tam_rewrite`			Lock for referencing[#3]	Update lock[#3]
`dc_tam_write`	For update		Lock for referencing[#2]	Update lock
	Either for update or addition or for addition		Update lock	--[#1]
`dc_dam_delete`			Update lock	--[#1]

Legend:

--: Not applicable

#1

Since the entire table is locked for updating, it is inaccessible to other transactions.

#2

A table of the reference type or the update type without permission for addition or deletion cannot be locked in this mode if "unlocked" is specified as the table lock mode in the TAM service definition.

#3

The resource is already made available by the function `dc_tam_read()` for update.

4.2.7 TAM table access facility without table-based lock

TP1/FS/Table Access 05-00 or earlier locks the appropriate individual tables when it adds or deletes records. This facility is referred to as the TAM table access facility with table-based lock. For information about the locking of individual tables, see *4.2.6 Lock for TAM tables*.

TP1/FS/Table Access 05-01 or later allows access to TAM table records while locking the appropriate records without locking entire tables. This facility is referred to as the TAM table access facility without table-based lock.

(1) How to use the TAM table access facility without table-based lock

To use the TAM table access facility without table-based lock, specify the update type that allows addition and deletion without locking tables in the TAM table access mode. The access mode must be specified in the `tamtable` command definition clause for the TAM service definition or the `tamadd` command. For information about the `tamtable` command definition clause, see the manual *OpenTP1 System Definition*. For information about the `tamadd` command, see the manual *OpenTP1 Operation*.

The same OpenTP1 system can include both TAM tables that are accessed using the TAM table access facility without table-based lock and TAM tables that are accessed using the TAM table access facility with table-based lock.

You do not need to recreate existing TAM files using the `tamcre` command before using the TAM table access facility without table-based lock.

(2) Lock

(a) Locking and unlocking resources

The `dc_tam_open` function and record access functions (`dc_tam_read`, `dc_tam_write`, and `dc_tam_delete`) lock resources. Similarly, UAPs written in COBOL and used to access records also lock resources.

Resources that have been locked are unlocked when the TAM table access transaction

ends.

(b) The lock setting of the TAM service function used to activate the TAM table access facility without table-based lock versus the actual lock status

The table below indicates the actual lock status, as compared to the lock setting of the TAM service function used to activate the TAM table access facility without table-based lock.

Table 4-10: Actual lock status, as compared to the lock setting of the TAM service function used to activate the TAM table access facility without table-based lock

Value assigned to the TAM service function and flag			Table-based lock	Record-based lock
dc_tam_open	Table-based lock		Update lock#1	--
	Record-based lock		--	--
dc_tam_read	Reference	No lock	--	--
		Lock applied	--	Reference lock
	Updating		--	Update lock
dc_tam_rewrite			--	Update lock#2
dc_tam_write			--	Update lock
dc_tam_delete			--	Update lock

Legend:

--: No lock is applied.

#1

A dc_tam_open function in which table-based lock is specified waits to lock the appropriate table if the table is already locked by another transaction's dc_tam_open function in which table-based lock is specified. However it does not wait for the completion of any function that accesses records rather than tables. For further information, see (3) in *4.2.7 TAM table access facility without table-based lock.*

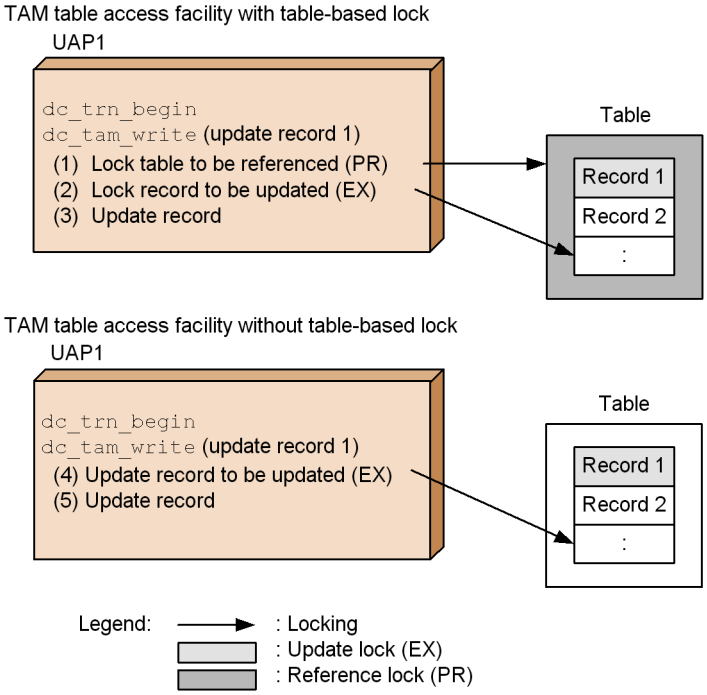
#2

The dc_tam_rewrite function does not lock resources, but the resources involved in this function have already been locked by the dc_tam_read function that was issued in update mode.

(c) Lock application processing

The figure below shows processing that is used to lock resources when updating records using the TAM table access facility with table-based lock and the TAM table access facility without table-based lock.

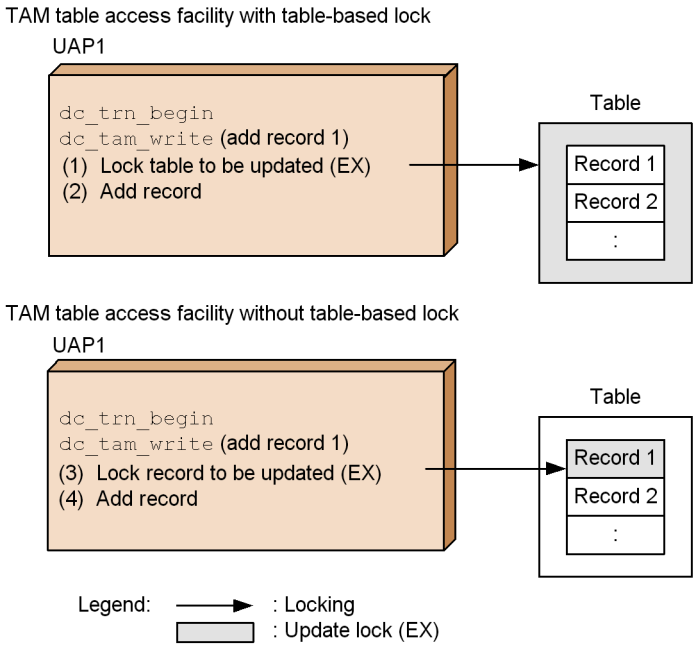
Figure 4-8: Locking resources when updating records



1. When `dc_tam_write` is issued, the TAM table access facility with table-based lock locks the table that is accessed for reference and locks the record to be updated. It then updates the record as shown in steps (1) to (3) in Figure 4-8.
2. When `dc_tam_write` is issued, the TAM table access facility without table-based lock locks the record to be updated, and then updates the record as shown in steps (4) and (5) in Figure 4-8.

The figure below shows processing that is used to lock resources when adding records using the TAM table access facility with table-based lock and the TAM table access facility without table-based lock.

Figure 4-9: Locking resources when adding records

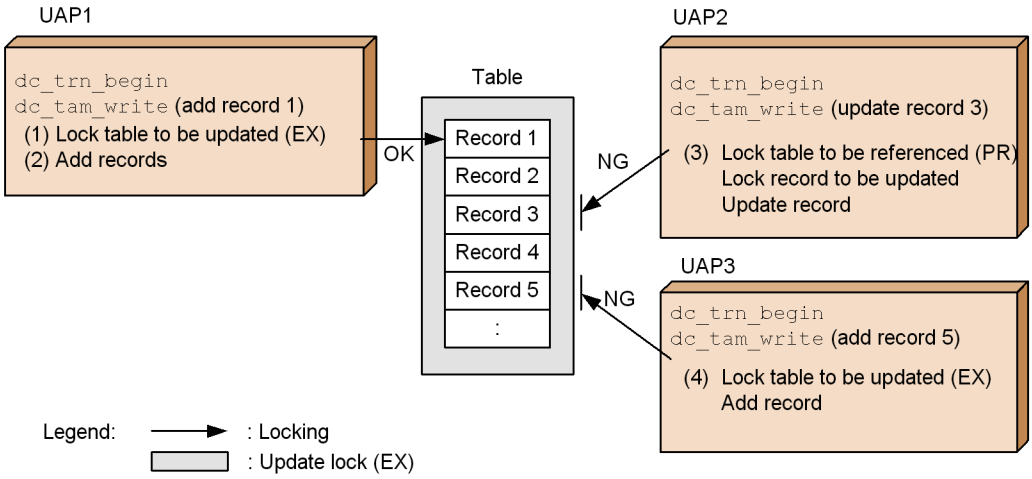


1. When `dc_tam_write` is issued, the TAM table access facility with table-based lock locks the table that is accessed for updating, and then adds a record as shown in steps (1) and (2) in Figure 4-9.
2. When `dc_tam_write` is issued, the TAM table access facility without table-based lock locks the record to be updated, and then adds the record as shown in steps (3) and (4) in Figure 4-9.

As explained above, the TAM table access facility with table-based lock and the TAM table access facility without table-based lock are different in the way they lock resources. For this reason, they are also different in the operation they will perform when two or more transactions compete for access to the same TAM table. If another transaction that is adding or deleting a record in the same table exists, the TAM table access facility with table-based lock cannot access the table to update, add, delete or reference the target record (if the table is locked by the function in the other transaction). The TAM table access facility without table-based lock can access the same TAM table if it does not compete with the facility in the other transaction for the record being accessed.

The figure below shows processing that is performed by the TAM table access facility with table-based lock when competition for access to the same record occurs.

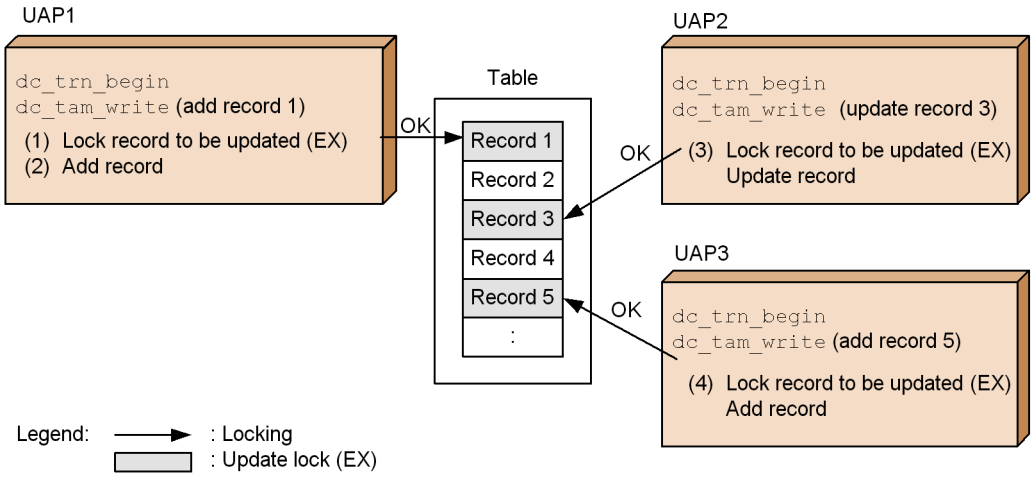
Figure 4-10: Processing that is performed by the TAM table access facility with table-based lock when competition for access to the same record occurs



1. UAP1 is going to add record 1. As shown in steps (1) and (2) in Figure 4-10, it locks the table to be updated, and then adds the record.
2. UAP2 cannot update record 3 because it cannot lock the table to be referenced, as shown at step (3) in Figure 4-10.
3. UAP3 cannot add record 5 because it cannot lock the table to be updated, as shown at step (4) in Figure 4-10.
4. Therefore, UAP2 and UAP3 will wait until UAP1 ends the transaction and unlocks the resource or UAP2 and UAP3 will abort with a `DCTAMER_LOCK` error.

The figure below shows processing that is performed by the TAM table access facility without table-based lock when competition for access to the same record occurs.

Figure 4-11: Processing that is performed by the TAM table access facility without table-based lock when competition for access to the same record occurs



1. UAP1 is going to add record 1. As shown in steps (1) and (2) in Figure 4-11, it locks record 1, and then adds the record.
2. UAP2 locks record 3 to be updated and then updates the record as shown at step (3) in Figure 4-11.
3. UAP3 locks record 5 to be added, and then adds the record as shown at step (4) in Figure 4-11.
4. In this way, UAP2 and UAP3 can access the same TAM table before UAP1 begins the transaction because UAP1 does not lock the table.

(3) Notes

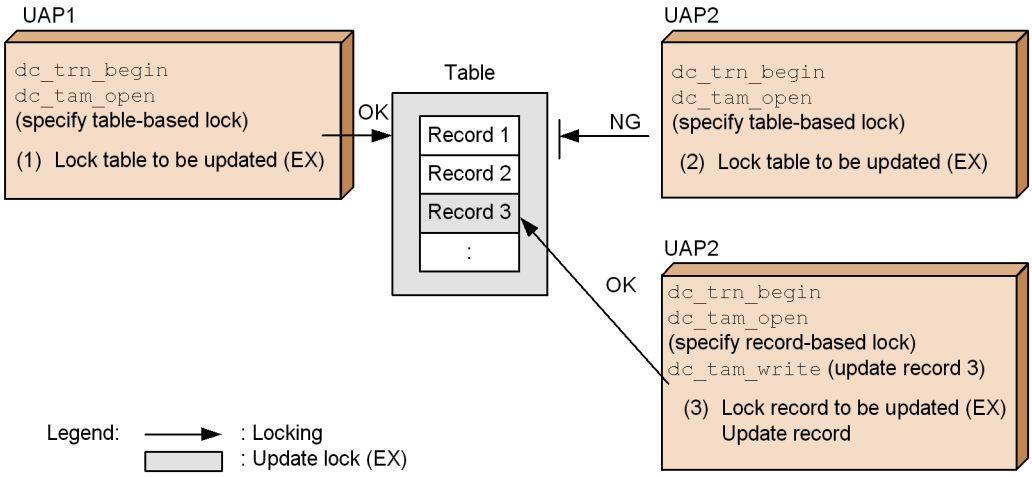
Note the following when using the TAM table access facility without table-based lock.

(a) Table-based lock applied by the dc_tam_open function

When the `dc_tam_open` function is issued with table-based lock specified (`DCTAM_TBL_EXCLUSIVE` `flags` is set to), it locks the table. However, it does not lock records that are included in the locked table and which may be accessed by record-access functions (`dc_tam_read`, `dc_tam_write`, and `dc_tam_delete`). This means that, once a `dc_tam_open` function that locks a table is issued, another `dc_tam_open` function that locks the same table must wait until the first function ends, but the record-access function does not need to wait.

The figure below shows how the `dc_tam_open` function locks resources.

Figure 4-12: How the dc_tam_open function locks resources



1. UAP1 issues a `dc_tam_open` function in which `flags` is set to `DCTAM_TBL_EXCLUSIVE`, in order to lock the table to allow updating as shown at step (1) in Figure 4-12.
2. UAP2 issues a `dc_tam_open` function in which table-based lock is specified. However, since it cannot lock the table to be updated as shown at step (2) in Figure 4-12, it will either wait until UAP1's transaction ends or abort with a `DCTAMER_LOCK` error.
3. UAP3 issues a `dc_tam_open` function in which record-based lock is specified (`flags` is set to `DCTAM_REC_EXCLUSIVE`). Therefore, the `dc_tam_open` function will end normally. UAP3 locks record 3 to be updated, and then updates the record as shown at step (3) in Figure 4-12.

(b) Allocating empty records when adding records

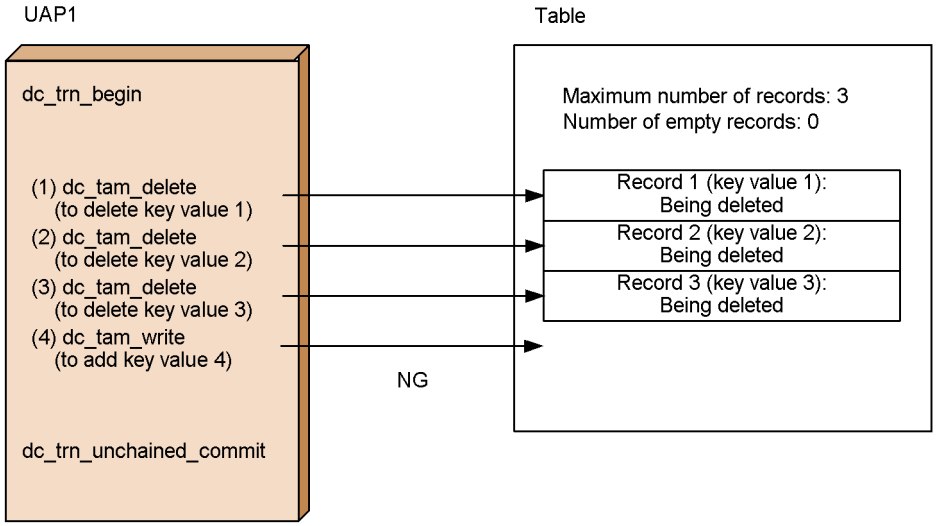
When a record is deleted, the record will not become empty until the transaction that has deleted the record is committed. This means that until the transaction is committed, the area reserved for the deleted record will not be allocated to records being added. However, when a record with the same key value as that of the deleted record is to be added within the same transaction that has deleted the record, the area for the deleted record is allocated for the record to be added.

Suppose that, in attempt to add records (even though there is not an equal number of empty records) you delete records with different key values within the same transaction that will add records. When you attempt to add records, you will encounter a `DCTAMER_NOAREA` error return.

The figure below shows how an attempt to add records causes a `DCTAMER_NOAREA`

error.

Figure 4-13: An example of a DCTAMER_NOAREA error caused by an attempt to add records

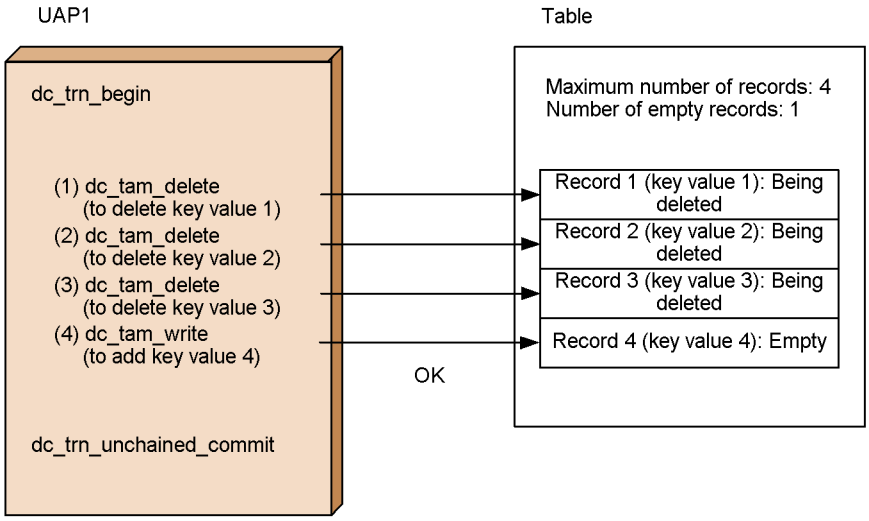


1. Assume that records with key values 1, 2, and 3 are stored in a TAM table that can contain up to 3 records. UAP1 deletes key values 1, 2, and 3 and adds key value 4.
2. As shown at step (1) in Figure 4-13, the deletion of key value 1 causes record 1 to be put into the deleted state, but does not cause it to become empty.
3. As shown at step (2) in Figure 4-13, the deletion of key value 2 causes record 2 to be put into the deleted state, but does not cause it to become empty.
4. As shown at step (3) in Figure 4-13, the deletion of key value 3 causes record 3 to be put into the deleted state, but does not cause it to become empty.
5. As shown at step (4) in Figure 4-13, an attempt to add key value 4 causes a `DCTAMER_NOAREA` error return because no empty record exists.

To prevent an attempt to add records from causing a `DCTAMER_NOAREA` error, you need to obtain a number of empty records equal to the number of records you want to add or wait until the transaction that deletes records is committed, and then add records.

The figure below shows processing that is performed to obtain a number of empty records equal to the number of records to be added.

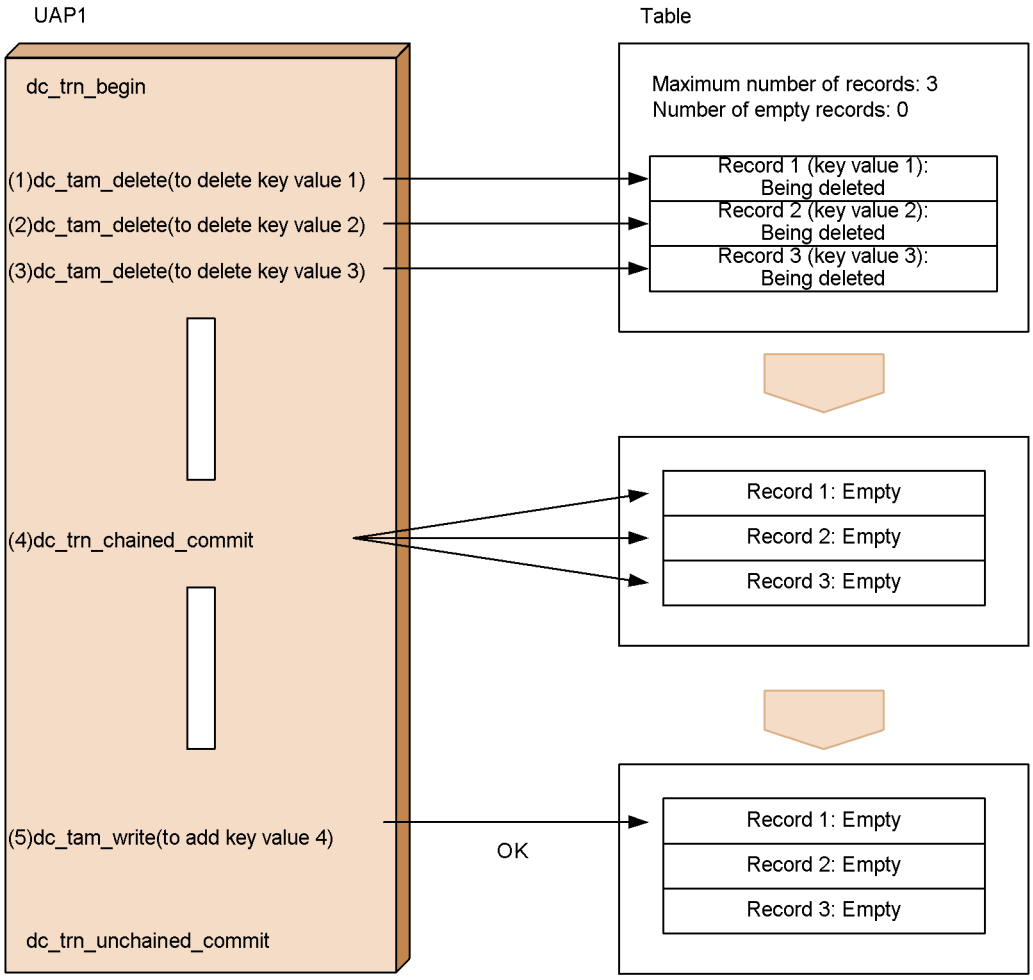
Figure 4-14: Processing that is performed to obtain a number of empty records equal to the number of records to be added



1. Increase the maximum number of records that can be stored in the TAM table, to 4. Assume that the TAM table contains records with key values 1, 2, and 3. UAP1 deletes key values 1, 2, and 3 and adds key value 4.
2. As shown at step (1) in Figure 4-14, the deletion of key value 1 causes record 1 to be put into the deleted state, but does not cause it to become empty.
3. As shown at step (2) in Figure 4-14, the deletion of key value 2 causes record 2 to be put into the deleted state, but does not cause it to become empty.
4. As shown at step (3) in Figure 4-14, the deletion of key value 3 causes record 3 to be put into the deleted state, but does not cause it to become empty.
5. As shown at step (4) in Figure 4-14, the addition of key value 4 allows an empty record 4 to be added.

The figure below shows processing that is performed to add records after the record deletion transaction is committed.

Figure 4-15: Processing that is performed to add records after the record deletion transaction is committed



1. Assume that records with key values 1, 2, and 3 are stored in a TAM table that can contain up to 3 records. UAP1 deletes key values 1, 2, and 3, commits the transaction, and then adds key value 4 during the next transaction.
2. As shown at step (1) in Figure 4-15, the deletion of key value 1 causes record 1 to be put into the deleted state, but does not cause it to become empty.
3. As shown at step (2) in Figure 4-15, the deletion of key value 2 causes record 2 to be put into the deleted state, but does not cause it to become empty.
4. As shown at step (3) in Figure 4-15, the deletion of key value 3 causes record 3 to be put into the deleted state, but does not cause it to become empty.

5. As shown at step (4) in Figure 4-15, the commitment of the transaction causes records 1, 2, and 3 being deleted to become empty.
6. As shown at step (5) in Figure 4-15, the addition of key value 4 allows an empty record 4 to be added.

(c) Access mode change

You cannot use the `tamadd` command to change a TAM table that uses the TAM table access facility with table-based lock to a TAM table that uses the TAM table access facility without table-based lock or vice versa. If you attempt to use the `tamadd` command in order to make such a change, the `tamadd` command ends abnormally.

If you want to switch the TAM table access facility with or without table-based lock to the other facility, change the `tamtable` command definition clause in the TAM service definition, and then start the OpenTP1 system as usual. Alternatively, start the OpenTP1 system, with no additional definition registered in the TAM service definition, and then use the `tamadd` command to add a new definition in the TAM service definition.

(d) Deadlock

A deadlock can occur when a TAM table using the TAM table access facility with table-based lock is changed to a TAM table that uses the TAM table access facility without table-based lock. For further information, see (1) (b) in *4.2.11 Notes on adding and deleting TAM records*.

(4) Programming interface

Except for the `dc_tam_status` function and `CBLDCTAM('INFO')`, TAM tables can be accessed via the same programming interface as for the TAM table access facility with table-based lock.

However, it may be necessary to recompile or relink UAPs. Table 4-11 lists conditions that require program recompilation. Table 4-12 lists conditions that require program relinkage.

Table 4-11: Conditions that require program recompilation

Condition				Work required
`dc_tam_status` used	Yes	`st_acs_type` referenced	Yes	A new constant `DCTAM_STS_RECLCK` is returned as access mode information. Therefore, you need to modify and recompile your UAPs.
			No	You do not need to recompile your UAPs.
	No			You do not need to recompile your UAPs.

Table 4-12: Conditions that require program relinkage

Condition		Work required
Libraries used by application programs	Archive libraries	You need to relink your UAPs
	Shared libraries	You do not need to relink your UAPs.

The dc_tam_status function returns access mode information to st_acs_type in the DC_TAMSTAT structure. Add DCTAM_STS_RECLCK as a value to be returned to st_acs_type. This value indicates an access mode in which records can be added or deleted without locking the table. A TAM table in this access mode uses the TAM table access facility without table-based lock.

For further information about the dc_tam_status function, its return values, and its usage, see the manual *OpenTP1 Programming Reference C Language*.

CBLDCTAM('INFO') returns access mode information to data name K. Add VALUE 'L' as a value to be returned to data name K. This value indicates an access mode in which records can be added or deleted without locking the table. A TAM table in this access mode uses the TAM table access facility without table-based lock. For further information about CBLDCTAM('INFO'), its return values, and its usage, see the manual *OpenTP1 Programming Reference COBOL Language*.

(5) Definition interface

In the tamtable command expression of the TAM service definition, the access type can be set using the -a option. When using the access facility for TAM tables without table-based locking, be sure to specify reclck as the -a option parameter. By setting this parameter to reclck, the "non-locking add/delete update type" access format is shown, which means that the TAM table is using the access facility for TAM tables without table-based locking.

For more detail on how to use other options of the tamtable command expression, see the manual *OpenTP1 System Definition.*

4.2.8 Creating TAM files

After allocating a direct file to the OpenTP1 file system, use the tamcre command of the commands to create a TAM file. At this time, specify an index type, a key area, and record data.

4.2.9 Interchangeability of TAM and DAM services

(1) DAM service functions able to access TAM tables

DAM file service functions (dc_dam_~) can be used to access TAM file records. In this case, a logical file name used for a DAM file is regarded as a TAM table name. A relative block number used for a DAM file is regarded as a TAM table key value. The following DAM service functions can be used to access TAM files:

- `dc_dam_open()` (Opens logical files.)
- `dc_dam_close()` (Closes logical files.)
- `dc_dam_read()` (Inputs logical file blocks.)
- `dc_dam_rewrite()` (Updates logical file blocks.)
- `dc_dam_write()` (Outputs logical file blocks.)

When the function `dc_dam_hold()` (which shuts down logical files) or the function `dc_dam_release()` (which releases logical files from the shutdown state) is issued for a TAM file, the function returns normally. However, the TAM file is not actually shut down or released from the shutdown state.

The following DAM service functions cannot be used to access TAM file records:

- All functions used for any job in offline mode
- `dc_dam_start()` (Start using an unrecoverable DAM file)
- `dc_dam_end()` (Terminate using an unrecoverable DAM file)
- `dc_dam_status()` (Reference the status of a logical file)

(2) TAM access by reading DAM file data

To enable TAM access for a DAM file, change the file as explained below:

1. Enter the DAM file data with the function `dc_dam_get()`, give a key value to each data item, then store the data times in any file.
2. Enter the file in 1, and execute the `tamcre` command to create a TAM file.

4.2.10 TAM service statistical information

Transaction statistical information which occurs when the TAM service is in use is acquired for individual transactions. Whether to output statistical information is determined by the value specified in the user service definition of the root transaction branch.

4.2.11 Notes on adding and deleting TAM records

To avoid deadlocks which occur as a result of attempts by UAPs to lock resources, you must regulate the type of lock and the lock sequence that each UAP implements for locking resources. This section explains how a deadlock occurs when UAPs lock resources for adding or deleting TAM records. It also gives advice for avoiding deadlocks.

(1) Cause of deadlock during a transaction

(a) When using the TAM table access facility with table-based lock

A TAM table which has not been locked for update may be updated or accessed more

than once during a single transaction. This type of transaction can cause a deadlock. A deadlock will occur in the following cases.

- A table is updated (records are added or deleted) or accessed (with lock specified) more than once during a single transaction.
- The access sequence for updating and accessing a TAM table is as follows:

  Access, then update.
- A TAM table is opened outside the transaction range or is opened inside the transaction range with record lock specified (this type of transaction is used in COBOL).
- Another transaction which needs to lock resources accesses the same table at the same time as the transaction described above.

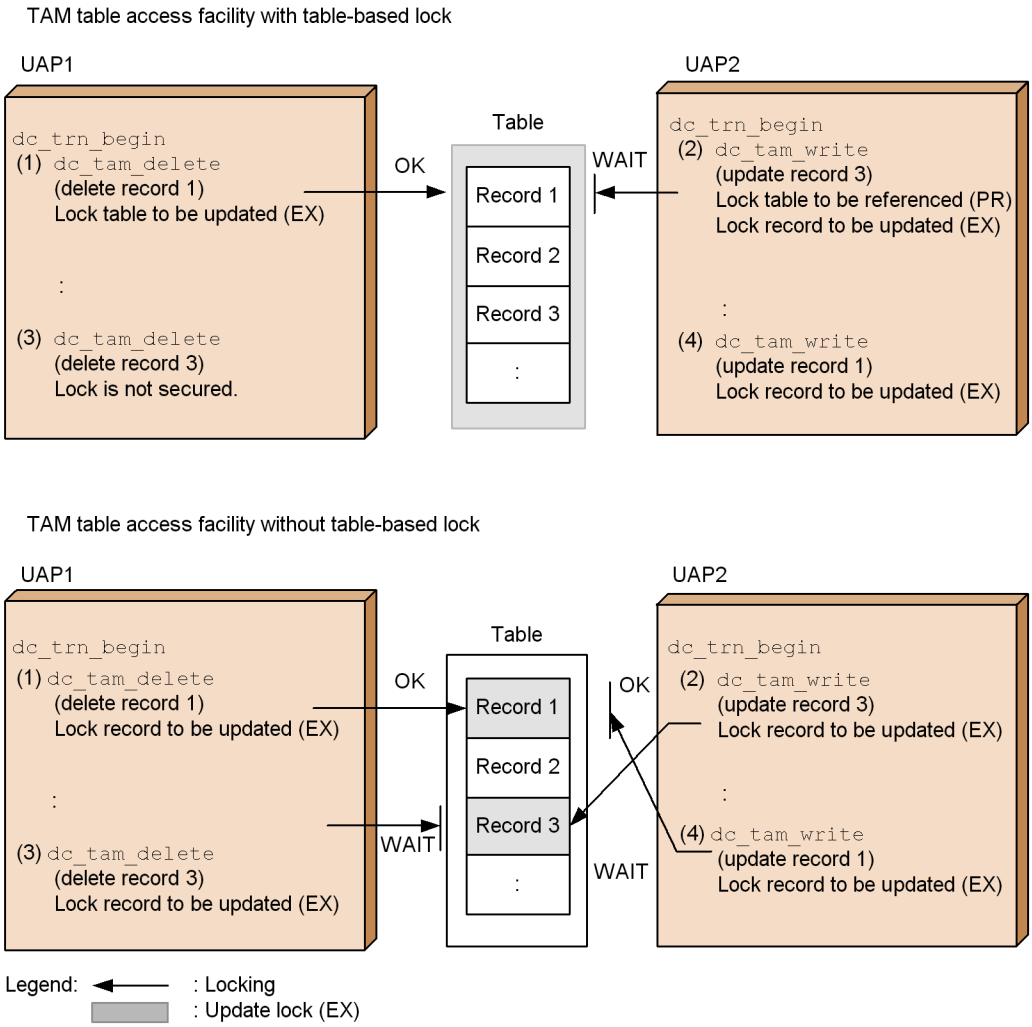
(b) When using the TAM table access facility without table-based lock

A deadlock may occur if you change a TAM table in which records were updated, added, or deleted using the TAM table access facility with table-based lock to a TAM table that uses the TAM table access facility without table-based lock.

When records are updated, added, or deleted using the TAM table access facility with table-based lock, the table is locked for table updating. Therefore, if two or more transactions are set to access the same records to be added or deleted, but in different sequences, no deadlock will occur because the transactions (except the one that is currently accessing records) wait because of table-based lock. However, a deadlock may occur if the records of the TAM table to be accessed are updated, added, or deleted using the TAM table access facility without table-based lock. Therefore, if you change a TAM table in which records were updated, added, or deleted using the TAM table access facility with table-based lock to a TAM table that uses the TAM table access facility without table-based lock, make the sequence in which the UAPs will access the records the same.

The figure below shows an example in which a deadlock occurs after change from a TAM table that uses the TAM table access facility with table-based lock to a TAM table that uses the TAM table access facility without table-based lock.

Figure 4-16: Example in which a deadlock occurs after change from a TAM table that uses the TAM table access facility with table-based lock to a TAM table that uses the TAM table access facility without table-based lock



Assume that UAP1 deletes records 1 and 3 in that order and that UAP2 updates records 3 and 1 in that order. Additionally, assume that the TAM table access facility with table-based lock and the TAM table access facility without table-based lock are performed in the sequence of (1) to (4) in the figure.

The TAM table access facility with table-based lock is performed in the following sequence:

1. To delete record 1, UAP1 locks the table for updating.
2. When UAP2 attempts to lock the table in order to update record 3, it encounters competition with the table lock at step 1. In order to lock the record for reference, it must wait until the table is unlocked.
3. When deleting record 3, UAP1 does not lock any resources.
4. When UAP1's transaction is committed, the table that was locked at step 1 is unlocked. UAP2 can now perform its processing.

   After UAP1's transaction is committed, UAP2 locks the reference table and locks the record to be updated, at step 2. At step 4, UAP2 locks record 1 to be updated.

The TAM table access facility without table-based lock is performed in the following sequence:

1. To delete record 1, UAP1 locks the record for updating.
2. To update record 3, UAP2 locks the record for updating.
3. When attempting to delete record 3, UAP1 encounters competition with the lock that was applied at step 2. Before locking the record to be updated, UAP1 must wait until the record is unlocked.
4. When attempting to update record 1, UAP2 encounters competition with the lock that was applied at step 1. Before locking the record to be updated, UAP2 must wait until the record is unlocked. A deadlock thus occurs.

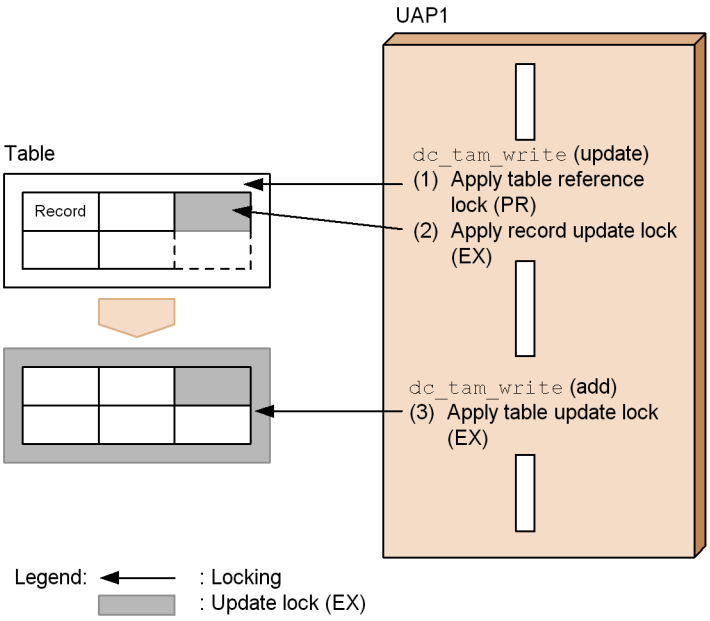
To avoid this deadlock, exchange steps 1 and 3 of UAP1 or exchange steps 2 and 4 of UAP2.

(2) Locking resources

The procedure by which resources are locked is explained below, accompanied by an example in which a resource is updated and added to. For information on locking a TAM table or record, see *4.2.6 Lock for TAM tables*.

The figure below shows an example of locking the resource to be updated and added to.

Figure 4-17: Example of update and addition



1. When the function `dc_tam_write()` is called for updating a resource, the resource is locked as shown in Figure 4-17.

   - Table reference lock (PR)
   - Record update lock (EX)

   The record to be updated is in a TAM table which has not been locked for update. The entire table is locked for reference to prevent another transaction from changing the record configuration within the table before the first transaction has ended.

2. When the function `dc_tam_write()` is called for adding to a resource, the resource is locked as shown in Figure 4-17.

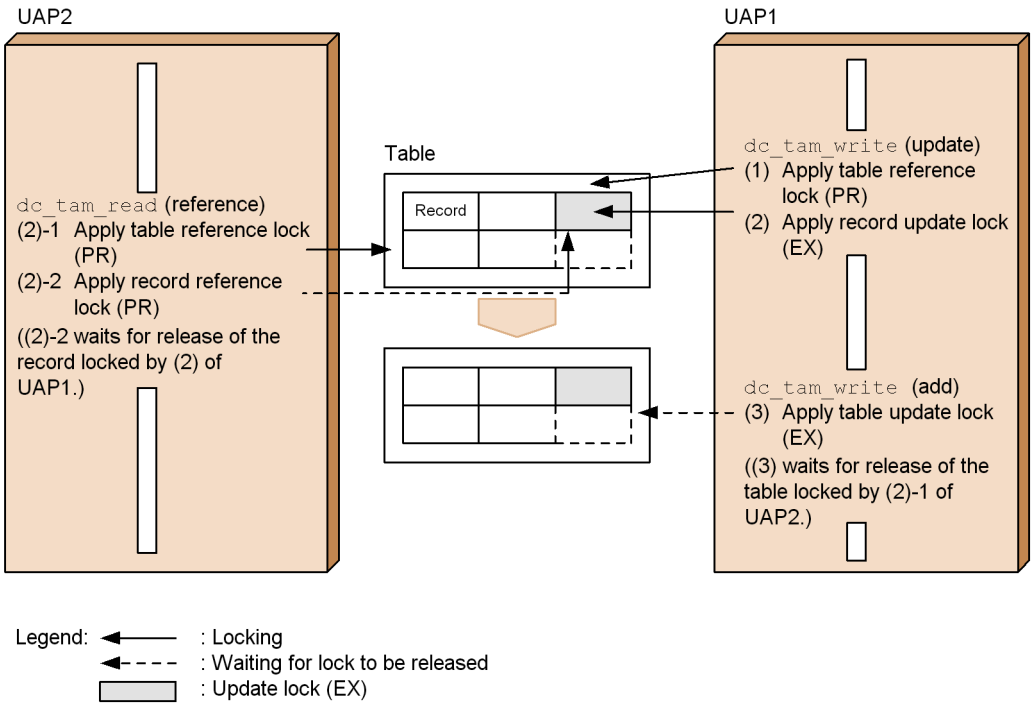
   - Table update lock (EX)

   Since this function changes the configuration within the table, the entire table is locked for update (EX) to prevent another transaction from referencing the table before the first transaction has ended.

3. Processing in steps 1 and 2 changes the lock on the table from a lock for reference (PR) to a lock for update (EX).

   The figure below shows how a deadlock occurs.

Figure 4-18: Occurrence of a deadlock



As shown at (2)-1 in Figure 4-18, another transaction can lock the table for reference (PR) after processing for step 1 has been completed but before processing for step 2 has started. This other transaction then attempts to lock a record updated by the first transaction. It waits for the lock on the record to be released while maintaining the lock for reference (PR) which it specified for the table ((2)-2 in Figure 4-18).

In processing for step 2, the first transaction cannot lock the table for update (EX) because the other transaction has locked the table for reference (PR). The first transaction waits for the lock to be released.

4. In step 3, both transactions wait for each other's resource to be freed. This situation is a deadlock.

   In Figure 4-18, a deadlock occurred because the local transaction continued processing (to the end of step 1) without specifying a lock for update (EX) for the resource (table) it required for addition processing. As a result, it allowed another transaction to lock the table. If the local transaction had specified a lock for update (EX) for the table in advance, the other transaction would not have been able to lock the table.

Make sure that transactions do not create a situation such as those described in (1) (a) above. For example, a table should not be updated or accessed more than once within a single transaction.

(3) Advice for avoiding deadlocks

If a TAM table must be updated or accessed more than once within a single transaction and a deadlock such as that described above is likely to occur, make sure that the transaction locks the table for update (EX) before it continues processing.

To lock a table for update, update (add or delete) the records first. Alternatively, if your program is in C, open the file so that it is locked within the transaction.

4.3 IST service (TP1/Shared Table Access)

The *IST service* is a facility which allows multiple OpenTP1 systems to share one or more tables across nodes. A table available with the IST service is called an *internode shared table*. An internode shared table can be used to reference or update its data from a UAP without recognizing the node at which the entity of the table exists. It can also be used as mail for managing the status of any job at each node. However, when data is distributed across multiple nodes, the IST service should not be used for the following jobs:

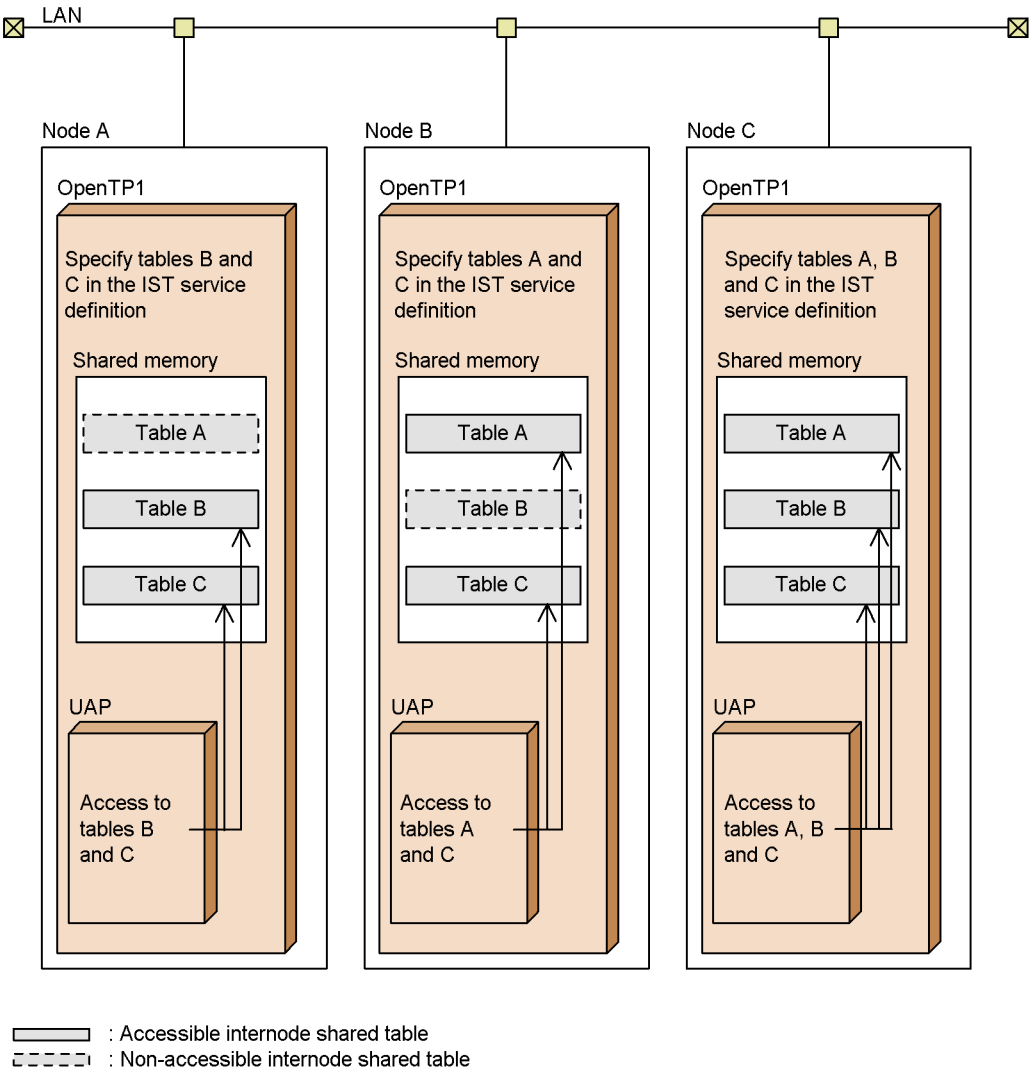
- Job that require immediate distribution of data
- Job that handles large amounts of data
- Job that updates data frequently

Before internode shared tables can be used, TP1/Shared Table Access must be installed in the system at each node. The IST service is available only when the basic OpenTP1 facilities are provided by TP1/Server Base. It is unavailable with TP1/LiNK.

4.3.1 System configuration of IST service

To use the IST service, for all nodes specify the same value in the internode shared table definition. If you do not specify the same value in the internode shared table definition for all nodes, the `KFCA25533-W` message will be output. The figure below shows an example when the same value is not specified in the internode shared table definition for all nodes.

Figure 4-19: When the same value is not specified in the internode shared table definition for all nodes



In Figure 4-19, the table names specified in the internode shared table definition for node A, node B, and node C are not the same. The system therefore considers that unexpected table information was received and it continues to periodically output the `KFCA25533-W` message on node A and node B until OpenTP1 terminates.

4.3.2 Outline of internode shared tables

This subsection outlines internode shared tables.

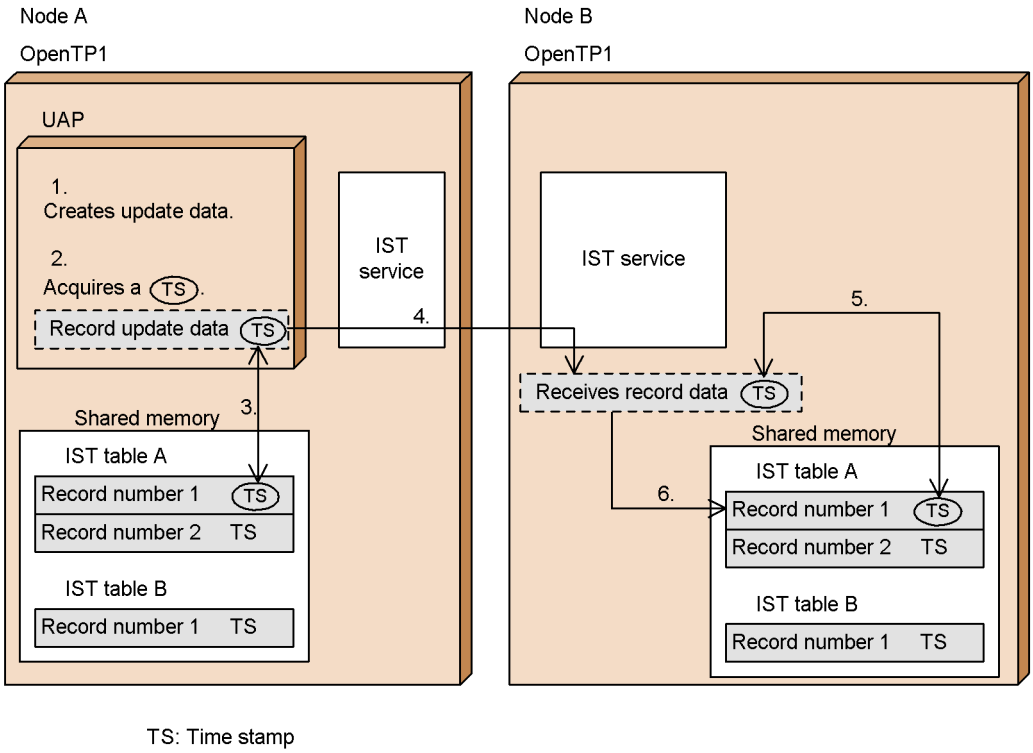
(1) Environment for access to internode shared tables

Internode shared tables reside in shared memory at each node. There is no file equivalent to the entity of each table. Therefore, internode shared tables can be accessed from a UAP in online environment only. They cannot be accessed in offline environment.

When the IST service is used across multiple nodes, the time must match among these nodes. If not, data updates at one node may not be reflected at another node.

The figure below shows the processing flow when the IST service updates an internode shared table record (a record contained within an internode shared table) on multiple nodes.

Figure 4-20: Updating an internode shared table record



1. The IST service creates record update data for updating an internode shared table record (record number 1) of the internode shared table A at node A.
2. It acquires the current time (machine time, in microseconds) and confers it as a time stamp on the record update data.

3. The IST service compares the time stamp set in the relevant internode shared table record in shared memory at node A and the time stamp given to the record update data.

   If the record update data is more recent, the IST service updates the internode shared table record in shared memory. If the record update data is older, the IST service does not update the internode shared table record in shared memory. Even when the IST service does not update the internode shared table record, the function `dc_ist_write()` returns normally.

4. When the IST service has updated the internode shared table record in shared memory, it notifies the IST service at node B that it has updated an internode shared table record at node A. At this time, it also reports the internode shared table record and the time stamp given to the internode shared table record.

5. The IST service at node B which received the updated internode shared table record compares the time stamp set in the relevant internode shared table record within the node and the time stamp of the internode shared table record that it received.

6. Only if the IST service determines as the result of step 5 that the time stamp of the internode shared table record that was received is more recent does it update the relevant internode shared table record at node B to the information provided in the internode shared table record that was received.

As explained above, the IST service determines whether to update an internode shared table record or leaves it as it is based on the time stamp. In the following cases, the latest update data may not be reflected in the internode shared table record.

- When the machine time at node A is later than the machine time at node B

  Sometimes after the IST service has updated an internode shared table record at node A, node B notifies the IST service that it has updated the same internode shared table record. Even in this case, the IST service regards the time stamp set in the internode shared table record at node A as being more recent. Therefore, the information in the updated internode shared table record at node B is not applied in the internode shared table record at node A.

  In addition, when the internode shared table record updated at node A is reported to node B, the IST service regards the time stamp of the reported internode shared table record as being more recent. Therefore, even if the corresponding internode shared table record at node B actually contains the latest information, it is updated to reflect the information contained in the reported internode shared table record.

- When the machine time at node A is earlier than the machine time at node B

  - When node B has updated an internode shared table record and the information in that internode shared table record has already been reported to node A

After an internode shared table record has been updated at node B, even if the IST service attempts to update the same internode shared table record at node A, the function `dc_ist_write()` returns normally without updating the record.

- When node B has updated an internode shared table record but the information in that internode shared table record has not yet been reported to node A

  After an internode shared table record has been updated at node B, when the IST service updates the same internode shared table record at node A, it updates the internode shared table record with the update information at node A. However, the IST service then regards the time stamp of the internode shared table record reported by node B to be more recent. This means that it reflects the information contained in the internode shared table record reported by node B in the internode shared table record at node A.

(2) Internode shared table structure

References and updates of an internode shared table from a UAP are done in units of records. An internode shared table consists of multiple records. A UAP process can access one record or access two or more records collectively.

4.3.3 Procedure for accessing an internode shared table

This subsection explains the procedure for accessing an internode shared table from a UAP. Access to an internode shared table cannot be committed or rolled back by using a transaction function.

(1) Opening internode shared tables

Before accessing an internode shared table from a UAP, the table must be opened first. To open the table, call the function `dc_ist_open()` [`CBLDCIST('OPEN')`]. When the table is opened, the table descriptor is returned. For processing after the table is opened, specify this table descriptor in the function to access the table. Keep the table descriptor in the UAP even for processing after the table is opened.

(2) Procedure for referencing/updating records

To input an internode shared table record, call the function `dc_ist_read()` [`CBLDCIST('READ')`]. To output data to an internode shared table record, call the function `dc_ist_write()` [`CBLDCIST('WRIT')`]. To call each of these functions, specify in its argument the table descriptor returned by the function `dc_ist_open()`.

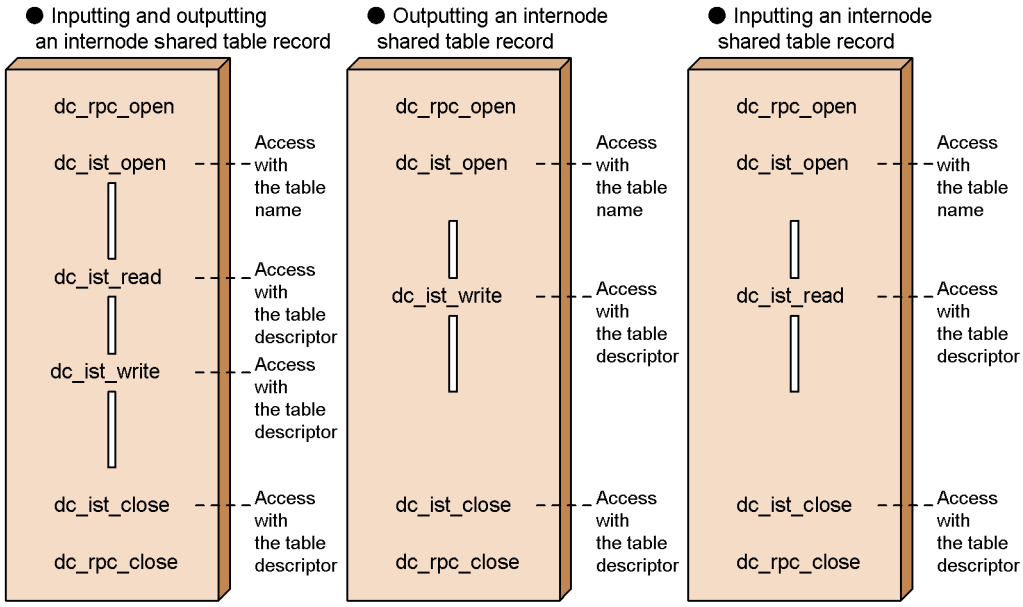
To input or output a record, the key values of more than one record can be specified collectively. A key value should be specified as a structure in the corresponding function. More than one structure can be specified.

(3) Closing internode shared tables

To close an internode shared table, call the function `dc_ist_close()` [`CBLDCIST('CLOS')`]. To call this function, specify in its argument the table descriptor returned by the function `dc_ist_open()`.

The figure below shows the procedures for accessing internode shared tables.

Figure 4-21: Procedures for accessing internode shared tables



4.3.4 Lock for internode shared tables

Internode shared tables are locked for each function called from a UAP. This lock control does not cause them to be occupied for the entire time after the data is entered and before it is updated. Therefore, even when one table is accessed from multiple UAPs, no deadlock occurs.

4.4 ISAM file service (ISAM, ISAM/B)

This section explains the ISAM file service for managing indexed sequential files. For details on this service, see the manual *Indexed Sequential Access Method ISAM*.

4.4.1 Outline of ISAM files

An indexed sequential file is composed of an index part for key management and a data file part for data storage. The use of the key allows sequential access and random access processing.

To manipulate ISAM files, library functions are called from UAPs or utility commands for ISAM file management are executed.

4.4.2 Types of ISAM service

OpenTP1 UAPs can use the following ISAM file services:

- ISAM
- ISAM/B

ISAM can be used in UAPs of both TP1/Server Base and TP1/LiNK. ISAM/B can be used only in UAPs of TP1/Server Base. ISAM/B cannot be used when the basic OpenTP1 facility is TP1/LiNK.

(1) ISAM

ISAM files are used as ordinary files. They are not synchronized with OpenTP1 transaction processing.

(2) ISAM/B

This facility allows the use of ISAM files in synchronization with transaction processing. If ISAM is used with ISAM/B, file integrity will be assured through transaction commitment/rollback.

(a) Products prerequisite to ISAM/B

Before ISAM files can be used with ISAM/B, the ISAM transaction facility (ISAM/B) must be available in addition to ISAM.

(b) Area for file creation

ISAM files to be used with ISAM/B must be created in the area allocated for the OpenTP1 file system.

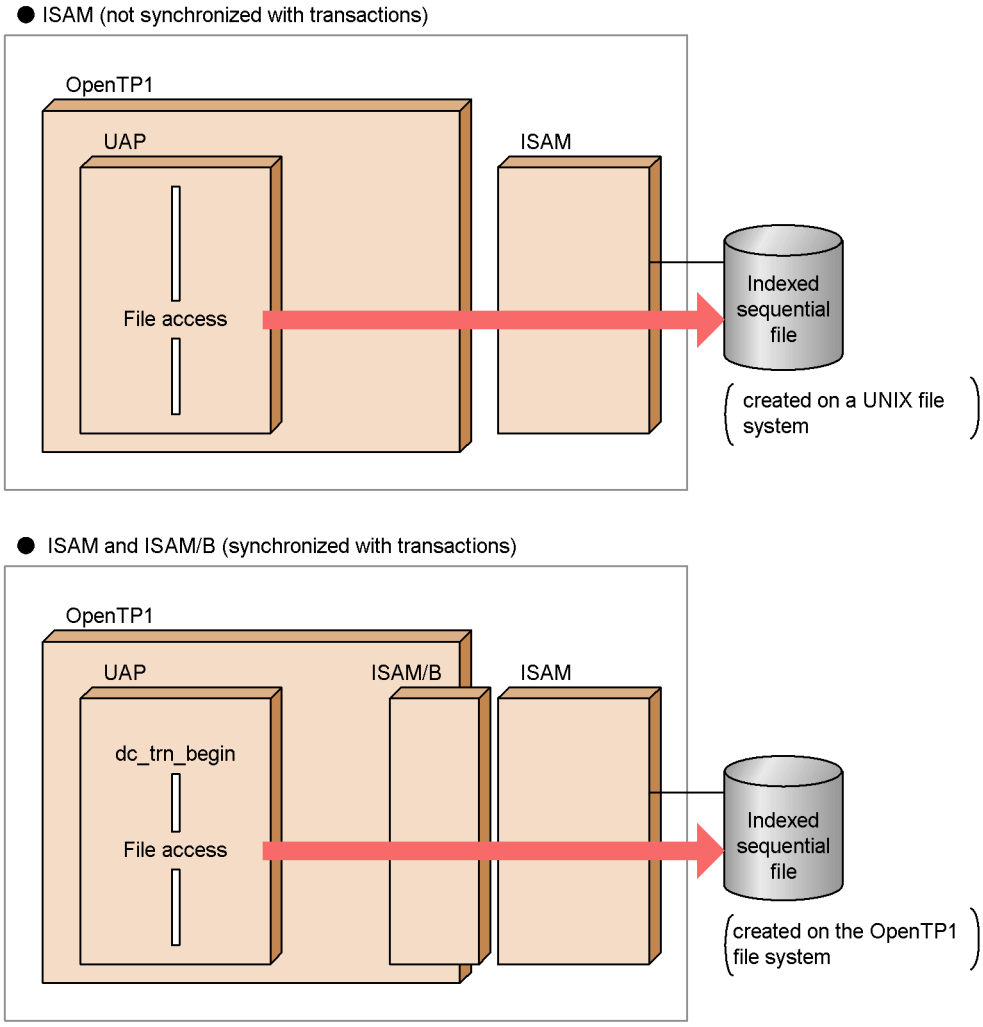
(c) Difference from OpenTP1 file service (TP1/FS/xxx)

ISAM/B does not use the lock service. Therefore, even when a deadlock occurs, the OpenTP1 lock service facility (such as lock scope reduction based on priority and

deadlock information output) is not available.

The figure below shows the form of ISAM file services.

Figure 4-22: Form of ISAM file services



4.5 Accessing database management systems

This section explains how database management systems (DBMSs) can be used in OpenTP1 UAPs.

4.5.1 Relation to OpenTP1 transaction processing

The usage of DBMSs depends on whether the DBMS supports the XA interface in the X/Open DTP model, and whether the DBMS can work with OpenTP1 transactions.

(1) DBMSs that support the XA interface

Only DBMSs that support the XA interface, for example ORACLE, can be controlled by OpenTP1 transaction processing. When a DBMS supports the XA interface, updates are possible using the commit and rollback operations of OpenTP1 transaction processing. In such transaction processing, you can use the functions that control OpenTP1 synchronization points (such as the functions `dc_trn_begin()`, `dc_trn_unchained_commit()`, `tx_begin()`, or `tx_commit()`). Facilities provided by a DBMS for controlling transactions cannot be used.

DBMSs that can be controlled through OpenTP1 transaction processing are limited to products supporting the XA interface.

In UAPs that access multiple databases, OpenTP1 allows updates while protecting the consistency of the multiple databases. The following OpenTP1 resource managers support the XA interface:

- TP1/FS/Direct Access (DAM file service)
- TP1/FS/Table Access (TAM file service)
- ISAM, ISAM/B (ISAM file service)
- TP1/Message Control (Message exchange facility (MCF))
- TP1/Message Queue (Message Queuing)

Thus, a UAP can process OpenTP1 transactions when accessing DBMSs that conform to the XA interface in the same way it does when it accesses the OpenTP1 resource manager. Even when some failure cause an abnormal termination of a UAP or when OpenTP1 is restarted, OpenTP1 performs a transaction determination (i.e., decides whether to perform a commit or rollback) for both the DBMS and the OpenTP1 resource manager.

(2) DBMSs that do not support the XA interface, or DBMSs that do not work with OpenTP1 via the XA interface

A DBMS that does not support the XA interface can be accessed, but cannot be synchronized with OpenTP1 transactions.

When a DBMS does not work with OpenTP1 via the XA interface, OpenTP1 cannot order a transaction determination to the DBMS in certain situations: such as when a UAP abnormally terminates during access to a database, or when OpenTP1 requires a rerun during access to a database. In such situations, you must recover the transactions using the DBMS facilities.

4.5.2 Preparation for using other vendors' DBMS in cooperation with OpenTP1 through XA interface

The following preparation is needed before another vendors' DBMS supporting the XA interface can be used in cooperation with OpenTP1 through the XA interface.

(1) Registration with OpenTP1

Register the names of various resource managers which are not provided by the OpenTP1. Use either methods to register them in OpenTP1:

- Use the `dcsetup` command to set up the OpenTP1, then execute the `trnlnkrm` command.
- Create an extended RM registration definition.

Once you create an extended RM registration definition, you need not execute the `trnlnkrm` command after setting up the OpenTP1 with the `dcsetup` command. For details on how to use the `trnlnkrm` command, see the manual *OpenTP1 Operation*. For details on how to specify extended RM registration definitions, see the manual *OpenTP1 System Definition.*

(2) UAP linkage

To create executable files for a UAP, you must link the object files used for transaction control with the DBMS libraries and object modules.

Use the `trnmkobj` command to make object files used for transaction control. For details on the `trnmkobj` command, see the manual *OpenTP1 Operation*.

(3) System definition

To use DBMSs, you must use trnstring in the transaction service definition and, if necessary, use trnrmid in the user service definition or user service default definition. Specified contents include the items for DBMSs. For details on such items, see the appropriate manuals for the database you use.

For details on definitions using trnstring and trnrmid, see the manual *OpenTP1 System Definition.*

When a non-OpenTP1 resource manager is used, you must define the set format in the transaction service definition and extend the size of the thread stack area.

For details on the set format definition, see the manual *OpenTP1 System Definition*.

(4) Environment variables

Some DBMS may require special environment variables for use with OpenTP1 UAPs. If they are necessary, you must use putenv in the transaction service definition, user service definition, or user service default definition.

For details on definitions using putenv, see the manual *OpenTP1 System Definition*.

4.6 Lock for resources

This section explains the method for allowing OpenTP1 UAPs to control locks of any user resources. To acquire a resource, it is necessary to call the function `dc_lck_get()` [`CBLDCLCK('GET')`] from the OpenTP1 UAP.

The facility for locking any resources is available only when the basic OpenTP1 facility is TP1/Server Base. It is unavailable with TP1/LiNK.

Locking of resources is used by processes which are being run as transactions. Since locks are specified for individual transactions, resources are correctly updated so that a particular UAP transaction process can exclusively update a resource at a given time.

For locking of DAM files, see *4.1 DAM file service (TP1/FS/Direct Access)*. For locking of TAM files, see *4.2 TAM file service (TP1/FS/Table Access)*.

4.6.1 Resources which can be put under lock

Resources (e.g., files) whose specific names have been defined in the operating system can be put under lock. Give a unique name in the node to a user-specific resource. OpenTP1 cannot judge if the name of the resource to be put under lock is correct. Specify the correct resource name in the UAP.

Lock can be specified in only the same node in an OpenTP1 system. Lock with a UAP in another OpenTP1 system cannot be specified.

4.6.2 Types of lock

To enable lock, specify a lock type (lock mode) and the resource-specific name in the OpenTP1 system. The following two lock modes are available:

Lock for reference (shared mode PR Protected Retrieve):

> The UAP can only reference resources with lock specified. Other UAPs are permitted only to reference the resources.

Lock for update (exclusive mode EX EXclusive):

> The UAP can reference and update resources with lock specified. Other UAPs are not permitted to reference or update the resources.

A resource cannot/can be shared depending on the contents of the lock modes in the following case:

- When an attempt is made to specify lock for the resource, lock has been specified for the resource by another UAP.

The table below shows whether a resource can be shared if lock has been specified for the resource by more than one UAP. If the resource cannot be shared, the following can be specified in the UAP:

- An error is returned, or the function waits until the resource is released.

Table 4-13: Combinations of lock modes and resource sharing enabled/disabled

Lock mode of UAP shutting down the resource	Lock mode of UAP requesting lock	
	Lock for reference (PR)	Lock for update (EX)
Lock for reference (PR)	Can be shared.	Cannot be shared.
Lock for update (EX)	Cannot be shared.	Cannot be shared.

4.6.3 Specifying the maximum lock wait time

If a UAP uses a lock request to the resource for which lock has been specified by another UAP, the UAP can wait until the resource is released. If more UAPs are waiting for the resource to be released, the order in which the UAPs will wait for the resource is decided according to the priority specified in the user service definition.

When the maximum lock wait time is specified in the lock service definition, and if a UAP waiting for the resource to be released waits for longer than the specified time, the UAP returns with an error.

The `lckls` command is provided for you to check the maximum lock wait time and the resource for which the UAP is waiting to be released.

4.6.4 Insufficient table pool for lock

Lock is managed in the table pool of the shared memory. If this table pool is full, an error is returned to the function that issued a resource lock request. In this case, use `abort()` with the service function in order to cancel lock processing.

4.6.5 Releasing a resource from lock

There are the following two methods for releasing a resource with lock specified:

- A resource is released from lock by the UAP shutting down the resource. To release a resource from lock by specifying the name of the resource, call the function `dc_lck_release_byname()` [`CBLDCLCK('RELNAME ')`]. To release all the resources shut down by the UAP at a time, call the function `dc_lck_release_all()` [`CBLDCLCK('RELALL  ')`].

The lock release functions can be called only from the UAP that specified lock for the resource. OpenTP1 allocates the resource released from lock to a UAP waiting for the resource.

- After the synchronization point processing of the UAP that shuts down the resource, OpenTP1 deallocates all the resources being shut down by the UAP. OpenTP1 automatically deallocates the resources regardless of whether the UAP terminates normally or abnormally.

4.6.6 Lock migration

If lock is specified for a resource by using the function `dc_lck_get()`, the resource lock right is transferred sequentially from one transaction branch to another in a global transaction. This facility is called *lock migration*. Lock migration prevents a deadlock or lock wait between transaction branches. Therefore, once lock is specified for a resource in a global transaction, the resource can be accessed from any transaction branch in the global transaction as long as the specification is in effect.

Lock migration is ensured in the following cases:

- The global transaction is in one node. (The global transaction does not comprise the services of multiple nodes.)

(1) Lock migration and lock modes

With lock migration, if the EX mode is specified in another transaction branch after lock has been specified in PR mode, all the subsequent lock is in EX mode. In a global transaction, once resources are put under lock in EX mode, the resource cannot be put under lock in PR mode. All resources are under lock in EX mode.

(2) Releasing resources with lock migration specified

If resources are under lock with lock migration specified, they are automatically released when the global transaction terminates. If the resources can be released before the termination of the global transaction, take the following procedures:

- Releasing resources with the function `dc_lck_release_byname()`

  To release a resource from lock with lock migration specified before commit/rollback processing is executed, specify the resource name, and call the function `dc_lck_release_byname()` (which releases resources from lock). The resource can be released from lock in any transaction branch. In this case, the resource cannot be released until the function `dc_lck_release_byname()` is called as many times as the lock count specified for the resource in the global transaction.

- Releasing resources with the function `dc_lck_release_all()`

  If the function `dc_lck_release_all()` (which releases all resources from lock) is called to a transaction branch, all the resources are released no matter how many times lock has been specified in any transaction.

(3) Notes on lock migration

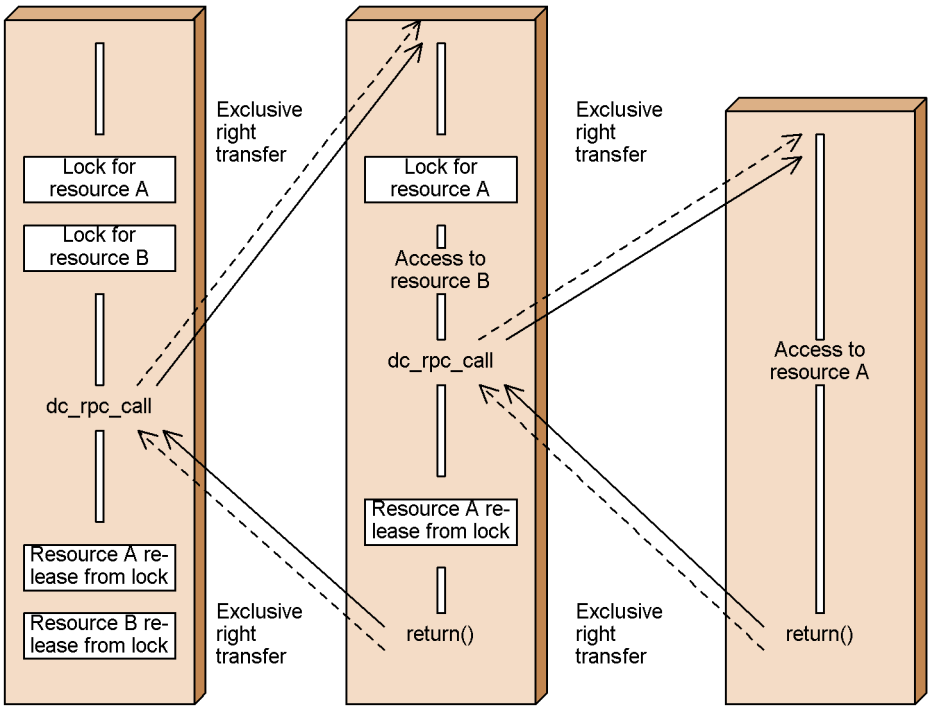
Do not access a resource for which lock has been specified (access to an already allocated resource or a new lock request) in the following case:

- After lock migration occurred with the function `dc_rpc_call()` of a synchronous-response-type RPC, the function `dc_rpc_call()` returned with an error (e.g., timeout).

Operation is not ensured if access is made to the resource.

The figure below shows lock migration.

Figure 4-23: Outline of lock migration



4.6.7 Lock test

A lock test (`DC_LCK_TEST` set in flags) is executed to decide whether lock can be specified for a resource by using the function `dc_lck_get()` (which puts resources under lock). In this case, the function `dc_lck_get()` terminates normally without putting the resource under lock (even when the resource can be put under lock). If the specified resource cannot be shared because it is lockly used by another UAP, the function `dc_lck_get()` returns with an error (`DCLCKER_WAIT` (00450)) regardless of whether waiting for release from lock is specified. Other return values for lock tests are listed below. Errors such as a deadlock, timeout, and insufficient memory are not returned. Lock migration is not generated.

- `DCLCKER_PARAM` (00401): The specified argument is invalid.
- `DCLCKER_OUTOFTRN` (00455): The function was called from a UAP outside the transaction processing range.
- `DCLCKER_VERSION` (00457): The OpenTP1 library version does not match the

lock service version.

(1) Notes on executing lock tests

Execute a lock test to check whether the resource can be put under lock. Even if a lock test terminates normally, this normal termination does not ensure that the subsequent lock requests will terminate normally. Also, even is a lock test terminates normally, the resource is not actually shut down. Thus, if the `dc_lck_release_all()` or `dc_lck_release_byname()` (which releases resources from lock) is called after the normal termination of a lock test, the function returns with an error.

4.7 Responses to the occurrence of deadlocks

OpenTP1 UAPs run in parallel while sharing resources with other UAPs. Each UAP locks resources so that there will be inconsistencies in changes to resources. However, if two or more UAPs attempt to acquire two or more resources in different sequences, they could stay inactive while waiting until each other's resource is freed. This condition is referred to as a *deadlock*.

In addition, if two or more UAPs attempt to access different resource managers (RMs), a deadlock could occur because file service lock control and TAM file service lock control influence each other. This section explains what is to be noted for avoiding deadlocks and also discusses OpenTP1 responses to deadlocks.

4.7.1 Notes for avoiding deadlocks

To avoid deadlocks, UAPs should access resources with the following considerations:

- If the UAP keeps a resource until the end of the transaction, it should acquire the resource as late as possible.
- A resource that can be freed during processing should be freed as soon as possible.
- If multiple resources are to be in use, the resource access sequences of UAPs should be standardized. In addition, the resource access sequences should be standardized within one system.
- The priorities for servicing UAPs which have encountered a deadlock should be predetermined.

4.7.2 OpenTP1 responses to deadlocks

If a deadlock occurs, OpenTP1 checks lock requests in terms of UAP lock wait priorities, selects those executed by UAP processes with lower priorities, and make the selected lock requests return with an error. The lock wait priority of a UAP is specified with `deadlock_priority` in the user service definition.

(1) UAP responses to deadlocks

If a function called in an attempt to acquire a resource returns with an error because of a deadlock, the UAP must do the following:

(a) Responses to deadlocks encountered during SUP or SPP processing

If a deadlock occurs during SUP or SPP processing, roll back the transaction using a rollback function (`dc_trn_unchained_rollback()`, `dc_trn_chained_rollback()`, or `tx_rollback()`). The SUP or SPP which is rolled back because of a deadlock is not retried. Reissue the request for the service from the client UAP.

(b) Responses to deadlocks encountered during MHP processing

If a deadlock occurs during MHP processing, call the function `dc_mcf_rollback()` to roll back the transaction. Whether to retry the MHP can be specified in the argument to the function `dc_mcf_rollback()`.

(2) Output of deadlock information and timeout information

If a deadlock occurs, detailed information about the UAP which caused the deadlock is output to the directory for the node containing the lock service. This information is called *deadlock information.*

Suppose that a UAP is waiting for the release of a resource. If the waiting interval exceeds the time specified for `lck_wait_timeout` in the lock service definition, the function called from the UAP returns with an error. Detailed information about the resource which was about to be acquired can be output to the directory of the node containing the lock service. This information is called *timeout information.*

Whether to output deadlock information and timeout information can be specified for `lck_deadlock_info` in the lock service definition.

For details on the output formats of deadlock information and timeout information, see *B. Output Format of Deadlock Information*.

(a) Deletion of deadlock information and timeout information

Deadlock information and timeout information can be deleted by either of the following methods:

- Delete the information using command

  Execute the `lckrminf` command.

- Delete the information which has been created in online mode when starting OpenTP1

  Specify the deletion conditions in the operands `lck_deadlock_info_remove` and `lck_deadlock_info_ remove_level` in the lock service definition.

(3) OpenTP1 responses to deadlocks involving multiple resource managers

If UAPs which are accessing multiple resource managers encounter a deadlock, OpenTP1 performs the following processing:

(a) Deadlock between RMs (DAM, TAM) which are lock-controlled by OpenTP1

The UAP lock wait priorities specified for `deadlock_priority` in the user service definition are observed when handling this type of deadlock.

(b) Deadlock between RM (DAM, TAM) which is lock-controlled by OpenTP1 and other vendors' RM

The lock waiting interval limit specified for `lck_wait_timeout` in the lock service definition is used for monitoring. Since RM-specific waiting interval limits are not referenced, do not forget to specify a lock waiting interval limit in the lock service definition.

(c) Deadlock between other vendors' RMs

Neither RM-specific waiting interval limits nor lock waiting interval limits as specified in the lock service definition are referenced. Instead, the transaction interval limit is used for monitoring UAPs. If the value given to `trn_expiration_time` in the user service definition, user service default definition, or transaction service definition is exceeded, the corresponding UAP process is terminated abnormally.

Chapter

5. X/Open-compliant Application Programming Interface

This chapter explains what facilities are available when the X/Open-compliant application programming interface (XATMI or TX) is used with OpenTP1 application programs.

The facilities are explained using C-language function names. For each function, the name of the equivalent COBOL-language API function is indicated in brackets [ ] when the function appears first in this chapter. After that, only the C-language function name is written. If the C-language function has no COBOL counterpart API function, brackets are not written.

This chapter contains the following sections:

5.1 XATMI interface (client/server-mode communication)
5.2 TX interface (transaction control)

5.1 XATMI interface (client/server-mode communication)

The XATMI interface is an application programming interface (API) which serves for client/server mode communication and conforms to the DTP model specified by X/Open, an open system standardization institute. OpenTP1 can use the XATMI interface for communication between UAP processes.

- Relationship between OpenTP1 UAP types and XATMI interface

  SUPs and SPPs can use the XATMI interface for communication. MHPs cannot use XATMI interface functions. To both the SUP and SPP, link stubs created from the XATMI interface definition file.

  As far as UAP process environments, methods for starting and termination, and OpenTP1 UAP operations are concerned, client/server mode communication through the XATMI interface is similar to client/server mode communication using OpenTP1 RPCs (`dc_rpc_call()`) unless otherwise specified.

5.1.1 Communication paradigms available with XATMI interface

This subsection explains communication paradigms available with the XATMI interface.

XATMI-interfaced communication uses TCP/IP as the communication protocol. Also, even if OSI TP is in use as the communication protocol, the XATMI interface can be used. For the relationship between communication protocols and XATMI interface functions, see *5.1.2 XATMI interface functions*.

OSI TP communication requires the TP1/NET/OSI-TP-Extended in the OpenTP1 system.

(1) Communication paradigms

The XATMI interface provides the following communication paradigms:

- Request/response service paradigm

  Communication based on this paradigm consists of sending one request to a service function and receiving one response. Like OpenTP1 remote procedure calls, communication is used to request services and receive results.

- Conversational service paradigm

  Communication based on this paradigm consists of activating the service function as the destination party and exchanging data to/from the service function via the connection established when the service function was activated.

  The conversational service paradigm is available only when TCP/IP is used as the communication protocol. If OSI TP is used as the communication protocol, the

conversational service paradigm is not available.

(2) Service request method

To request a service, use a function whose argument specifies a name (service name) that identifies a service function in the server UAP.

(3) Data that can be sent and received with XATMI-interfaced communication

When XATMI-interfaced communication is in use, structures in C or records in COBOL can be transmitted and received. This means that a chunk of data of some size can be sent with a single service request. Such a chunk of data is referred to as a *typed buffer* in C or a *typed record* in COBOL. For typed data used in communication, see *5.1.6 Communication data types*.

5.1.2 XATMI interface functions

This subsection explains the XATMI interface facilities which are available with each application programming interface under each communication protocol.

(1) XATMI interface library functions

Table 5-1 lists the XATMI interface library functions.

Table 5-1: XATMI interface library functions

XATMI interface facilities		Library function name	
		C language library	COBOL language library
Request/ response service paradigm	Send a service request and synchronously await its reply	`tpcall()`	`TPCALL`
	Send a service request	`tpacall()`	`TPACALL`
	Get a reply from a previous service request	`tpgetrply()`	`TPGETRPLY`
	Cancel a call descriptor for an outstanding reply	`tpcancel()`	`TPCANCEL`
Conversational service paradigm	Establish a conversational service connection	`tpconnect()`	`TPCONNECT`
	Terminate a conversational service connection abortively	`tpdiscon()`	`TPDISCON`
	Receive a message in a conversational connection	`tprecv()`	`TPRECV`
	Send a message in a conversational connection	`tpsend()`	`TPSEND`

XATMI interface facilities		Library function name	
		C language library	COBOL language library
Manipulation of communication data types	Allocate a typed buffer	tpalloc()	--
	Free a typed buffer	tpfree()	--
	Change the size of a typed buffer	tprealloc()	--
	Determine information about a typed buffer	tptypes()	--
Dynamic management of service names	Advertise a service name	tpadvertise()	TPADVERTISE
	Unadvertise a service name	tpunadvertise()	TPUNADVERTISE
Functions used by server	Template for (entity of) service routines#	tpservice()	--
	Start a service routine#	--	TPSVCSTART
	Return from a service routine	tpreturn()	TPRETURN

Legend:

--: There is no counterpart API function.

#

Since the method for declaring the start of a service is different between C and COBOL, separate API components are provided in these languages. The function tpservice() denotes the entity of the service in C.

Table 5-2 gives the relationship between XATMI interface functions and OpenTP1 UAPs.

Table 5-2: Relationship between XATMI interface functions and OpenTP1 UAPs

XATMI interface function	SUP		SPP			MHP		Off-line
	Outside	Inside	Outside	Transaction processing range		Outside	Inside	
				Root	Not root			
tpacall()	Y	Y	Y	Y	Y	--	--	--
tpadvertise()	--	--	Y#1	Y#1	Y#1	--	--	--

XATMI interface function	SUP		SPP			MHP		Off-line
	Outside	Inside	Outside	Transaction processing range		Outside	Inside	
				Root	Not root			
tpalloc()	Y	Y	Y	Y	Y	--	--	--
tpcall()	Y	Y	Y	Y	Y	--	--	--
tpcancel()	Y	Y	Y	Y	Y	--	--	--
tpconnect()	Y	Y	Y	Y	Y	--	--	--
tpdiscon()	Y	Y	Y	Y	Y	--	--	--
tpgetrply()	Y	Y	Y	Y	Y	--	--	--
tpfree()	Y	Y	Y	Y	Y	--	--	--
tprecv()	Y	Y	Y	Y	Y	--	--	--
tprealloc()	Y	Y	Y	Y	Y	--	--	--
tpreturn()	--	--	Y#2	Y#2	Y#2	--	--	--
tpsend()	Y	Y	Y	Y	Y	--	--	--
tpservice()#3	--	--	--	--	--	--	--	--
tptypes()	Y	Y	Y	Y	Y	--	--	--
tpunadvertise()	--	--	Y#1	Y#1	Y#1	--	--	--

Legend:

Outside: Outside transaction range

Inside: Inside transaction range (root)

Off-line: UAP that handles offline work

Y: The function can be used with UAPs.

--: The function cannot be used with UAPs.

Note

The *outside the transaction range* for MHP means the range of MHPs with the nontransaction attribute or the main function of MHPs.

#1

Can be called only within service functions.

#2

Used only to make XATMI-interfaced service functions return.

#3

`tpservice()` is the entity of the service function.

(2) Relationship between XATMI interface facilities and communication protocols

The XATMI interface can be used for both TCP/IP communication and OSI TP communication. However, there may be restriction on some facilities depending on the communication protocol. Table 5-3 gives the relationship between XATMI interface facilities and communication protocols.

Table 5-3: Relationship between XATMI interface facilities and communication protocol

XATMI interface facility	Communication protocols	
	TCP/IP	OSI TP
Request/response service paradigm	Y	Y
Conversational service paradigm	Y	--
Typed data transmission	Y	Y#
Client/server mode communication between OpenTP1 systems	Y	Y
Transaction extension to other TP monitors	--	Y

Legend:

Y: Can be used under this communication protocol.

--: Cannot be used under this communication protocol.

#

When OSI TP is used for client/server mode communication with non-OpenTP1 system, data can be transmitted if its type is appropriately converted. For the communication data types that can be specified, see *5.1.6 Communication data types*.

5.1.3 Request/response service paradigm

XATMI-interfaced communication based on the request/response service paradigm is explained below.

(1) Types of request/response service

The following types of request/response service based communication are available:

(a) Communication with synchronous response reception

A request for a service is issued, then a response is awaited. The function `tpcall()` `[TPCALL]` is used to request a service.

Time monitoring:

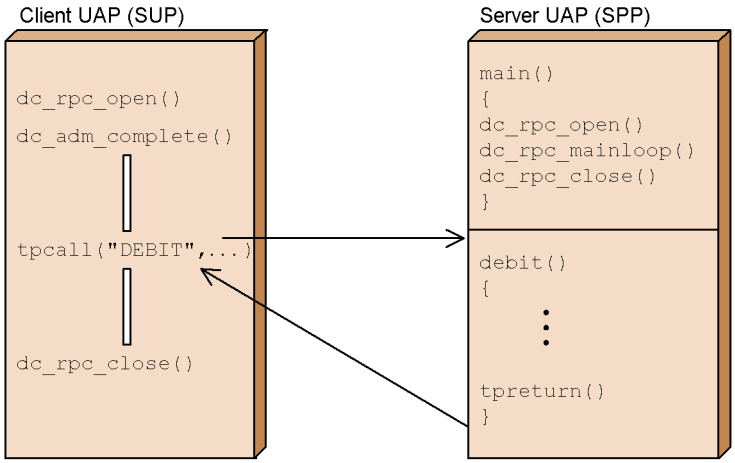
Communication with synchronous response reception involves monitoring of times taken before a response returns. The maximum response waiting interval is specified in the OpenTP1 definition. For details on the definition, see the manual *OpenTP1 System Definition*.

If the process which calls the function `tpcall()` is under a transaction, the maximum response waiting interval is the value assigned to `trn_expiration_time` in the definition. In this case, the process terminates abnormally when the maximum response waiting interval expires (`tpcall()` does not return with an error).

If the process which calls the function `tpcall()` is not under a transaction, the maximum response waiting interval is the value assigned to `watch_time` in the definition. In this case, the function `tpcall()` returns with an error when the maximum response waiting interval expires.

The figure below shows the communication with synchronous response reception based on the request/response service paradigm.

Figure 5-1: Communication with synchronous response reception based on request/response service paradigm



(b) Communication with asynchronous response reception

A service is requested, then processing continues without waiting for a response. Then, a function is issued to receive a response. The function `tpacall()` [`TPCALL`] is used to request service and the function `tpgetrply()` [`TPGETRPLY`] is used to receive a response.

Time monitoring:

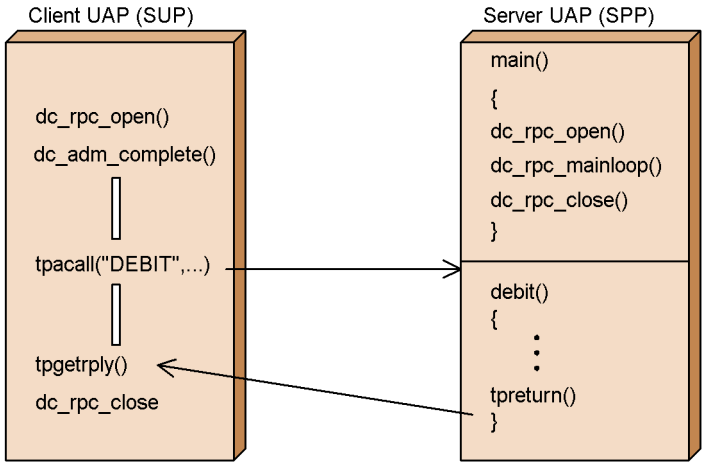
Communication with asynchronous response reception involves a process in which the communication party waits until the function `tpgetrply()` receives a response. The maximum response waiting interval is specified in the OpenTP1 definition. For details on the definition, see the manual *OpenTP1 System Definition.*

If the process which calls the function `tpacall()` or `tpgetrply()` is under a transaction, the maximum response waiting interval is the value assigned to `trn_expiration_time` in the definition. In this case, the process terminates abnormally when the maximum response waiting interval expires (`tpgetrply()` does not return with an error).

If the process which calls the function `tpacall()` or `tpgetrply()` is not under a transaction, the maximum response waiting interval is the value assigned to `watch_time` in the definition. In this case, the function `tpgetrply()` returns with an error when the maximum response waiting interval expires.

The figure below shows the communication with asynchronous response reception based on the request/response service paradigm.

Figure 5-2: Communication with asynchronous response reception based on request/response service paradigm



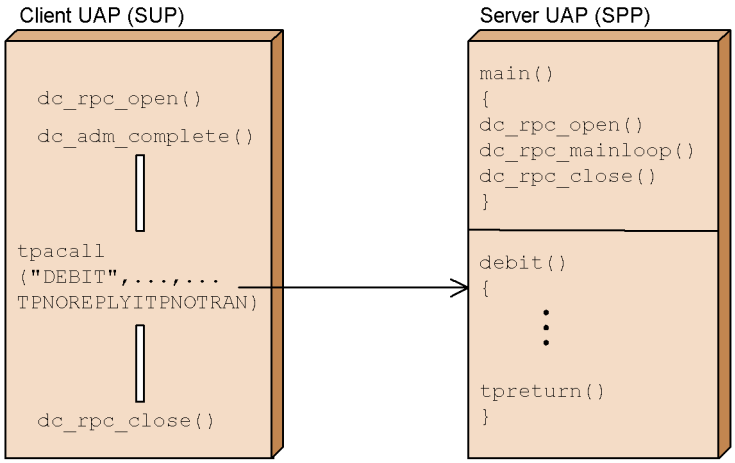
(c) Communication without response reception

The result of the service request is not returned. With this type of communication, `TPNOREPLY` is specified for flags of the function `tpacall()`. Note that `TPNOTRAN` must also be specified for flags of the function `tpacall()`.

When a service request is used through this type of communication, no response is returned. The UAP which requested that service continues processing.

The figure below shows the communication without response reception based on the request/response service paradigm.

Figure 5-3: Communication without response reception based on request/response service paradigm



(2) Time monitoring involved in communication based on request/response service paradigm

In communication based on the request/response service paradigm, time monitoring is always performed according to the values specified in the OpenTP1 definition. For details on the definition, see the manual *OpenTP1 System Definition*.

There are two types of timeout: transaction timeout and blocking timeout. The transaction timeout is a timeout that is encountered during the transaction process, whereas the blocking timeout is caused by a wait in blocking status. If a transaction timeout occurs, the process terminates abnormally.

The process may wait beyond the specified timeout value (because the OpenTP1 monitoring timer is reset even if a response other than the appropriate data is returned). If `TPNOTIME` is specified for flags, the timeout value is infinite. Note that the transaction timeout occurs regardless of whether this flag is set.

(3) Relationship between communication based on request/response service paradigm and transaction

The transaction is controlled by a transaction control function available with OpenTP1 (function `dc_trn_~` or `tx_~`). Whether a transaction under OpenTP1 is settled or is in `rollback_only` status is determined from the result of the service function or the transaction control function. In communication based on the request/response service paradigm, however, the transaction branch concerned enters `rollback_only` status if one of the following errors occurs:

- Transaction timeout (the process terminates abnormally)
- typed buffer reception error (an illegal type was received)

- TPESVCERR or TPESVCFAIL error (this error is reported by the function tpreturn() or by user with the function tpreturn())
- TPESYSTEM error (the transaction may not enter rollback_only status even if TPESYSTEM returns)

(a) Transaction control involved in communication with synchronous response reception

If the caller process is under a transaction, whether the called service is treated as a transaction is determined by the parameter set in the flags argument to the function tpcall(). Set TPNOTRAN only when the called service is not to be treated as a transaction. A transaction timeout may occur even if TPNOTRAN is set.

(b) Transaction control involved in communication with asynchronous response reception

If the caller process is under a transaction, whether the called service is treated as a transaction is determined by the parameter set in the flags argument to the function tpacall(). Set TPNOTRAN only when the called service is not to be treated as a transaction. A transaction timeout may occur even if TPNOTRAN is set.

(c) Transaction control involved in communication without response reception

In communication without response reception, no service can be treated as a transaction. TPNOTRAN must always be specified for the flags argument to the function tpacall().

(4) Responses to blocking

Functions for communication based on the request/response service paradigm have the TPNOBLOCK flag that indicates the necessary action against blocking. The function tpgetrply() with this flag returns with an error if it detects blocking. If this flag is not set, the function waits until blocking is removed or a blocking timeout occurs (if TPNOTIME is set, however, no blocking timeout error occurs). Under OpenTP1, this flag has effect only on the function tpgetrply(). Even if this flag is set for the function tpcall() or tpacall(), it is ignored.

5.1.4 Conversational service paradigm

This subsection explains the conversational service paradigm available with the XATMI interface. Communication based on this paradigm is possible only when the communication protocol is TCP/IP. If OSI TP is used as the communication protocol, the conversational service paradigm is not available.

(1) Procedure for communication based on conversational service paradigm

Communication based on this paradigm is started after a connection is established with the destination party through the issuance of a function. The procedure for requesting

service is as follows:

(a) Connection establishment

The client UAP establishes a connection with the service function by the function `tpconnect()` [`TPCONNECT`]. The UAP process that established the connection by using the function `tpconnect()` is referred to as the *originator* and the remote UAP process is referred to as the *subordinator*.

When the function `tpconnect()` returns normally, a positive descriptor that identifies the established connection is returned. This descriptor is assigned to the communication function and communication starts.

When the function `tpreturn()` is called by the service function to terminate the process, the connection is terminated.

Connection control authority:

Either `TPSENDONLY` or `TPRECVONLY` that indicates whether the process has control authority should be specified for the `flags` argument to the function `tpconnect()`. If `TPSENDONLY` is specified, the process acquires control authority so that it can call the data sending function `tpsend()`. In contrast, if `TPRECVONLY` is set, control authority is passed to the remote process so that the caller process of the function `tpconnect()` can call the data reception function `tprecv()`.

(b) Data sending (tpsend())

To send data, call the function `tpsend()` `[TPSEND]`. Set the descriptor returned by the function `tpconnect()` for the argument to the function `tpsend()` so that the connection to be used can be identified.

No communication party can call the function `tpsend()` if it has not connection control authority. Even if called, the function `tpsend()` returns with an error.

To pass connection control authority to the remote process, specify `TPRECVONLY` for flags of the function `tpsend()`. By using the function `tpsend()` with this flag specified, control authority is passed to the remote process.

To send data from the subordinator with the function `tpsend()`, use the descriptor obtained from the received `TPSVCINFO` structure.

(c) Data receiving (tprecv())

To receive data, call the function `tprecv()` [`TPRECV`]. Data is received asynchronously. The function `tprecv()` can be issued from processes that have no control authority.

Time monitoring:

If `TPNOBLOCK` is not specified for flags, the function `tprecv()` waits until data is received. The maximum response waiting interval is specified in the OpenTP1

definition. For details of the definition, see the manual *OpenTP1 System Definition*.

If the process which calls the function `tprecv()` is under a transaction, the maximum response waiting interval is the value assigned to `trn_expiration_time` in the OpenTP1 system definition. In this case, the process terminates abnormally when the maximum response waiting interval expires (`tprecv()` does not return with an error).

If the process which calls the function `tprecv()` is not under a transaction, the maximum response waiting interval is the value assigned to `watch_time` in the OpenTP1 system definition. In this case, the function `tprecv()` returns with an error when the maximum response waiting interval expires.

(d) Disconnection

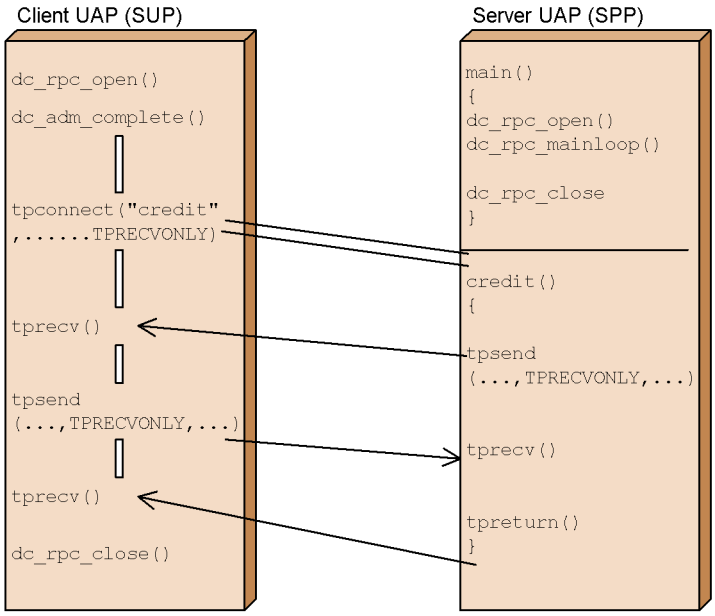
A connection is terminated normally when processing of the service function is completed and the function `tpreturn() [TPRETURN]` is called with appropriate control authority. A connection may also be terminated due to a communication error.

(e) Forced disconnection (tpdiscon())

To forcibly terminate a connection for some reason, call the function `tpdiscon() [TPDISCON]`. The descriptor specified in the function `tpdiscon()` has no effect on the subsequent processes. For transaction processing, the transaction branch on the subordinator enters `rollback_only` status.

The figure below shows the communication based on the conversational service paradigm.

Figure 5-4: Communication based on conversational service paradigm



(2) Time monitoring involved in communication based on conversational service paradigm

In communication based on the conversational service paradigm, time monitoring is always performed according to the values specified in the OpenTP1 definition. For details on the definition, see the manual *OpenTP1 System Definition*.

There are two types of timeout: transaction timeout and blocking timeout. The transaction timeout is a timeout that is encountered during the transaction process, whereas the blocking timeout is caused by a wait in blocking status. If a transaction timeout occurs, the process terminates abnormally.

The process may wait beyond the specified timeout value (because the OpenTP1 monitoring timer is reset even if a response other than the appropriate data is returned). If `TPNOTIME` is specified for flags, the timeout value is infinite. Note that the transaction timeout occurs regardless of whether this flag is set.

(3) Relationship between communication based on conversational service paradigm and transaction

The transaction is controlled by a transaction control function available with OpenTP1 (function `dc_trn_~` or `tx_~`). Whether the transaction under OpenTP1 is settled or in `rollback_only` status is determined from the result of the service function or the transaction control function. In communication based on the conversational service paradigm, however, the transaction branch concerned enters `rollback_only` status

if one of the following errors occurs:

- Transaction timeout (the process terminates abnormally)
- typed buffer reception error (an illegal type was received)
- TPESYSTEM error (the transaction may not enter rollback_only status even if TPESYSTEM returns)
- Calling of tpdiscon()
- Error code TPEEVENT returned by the function tpsend() (event code is TPEV_SVCERR or TPEV_SVCFAIL)
- Error code TPEEVENT returned by the function tprecv() (event code is TPEV_DISCONIMM, TPEV_SVCERR, or TPEV_SVCFAIL)

If the caller process is under a transaction, whether the called service is treated as a transaction is determined by the parameter set in the flags argument to the function tpconnect(). Set TPNOTRAN only when the called service is not to be treated as a transaction. A transaction timeout may occur even if TPNOTRAN is set.

(4) Responses to blocking

Functions for communication based on the conversational service paradigm have the TPNOBLOCK flag that indicates the necessary action against blocking. The function tprecv() with this flag returns with an error if it detects blocking. If this flag is not set, the function waits until blocking is removed or a timeout occurs (if TPNOTIME is set, however, no blocking timeout error occurs). With OpenTP1, this flag has effect only on the function tprecv(). Even if this flag is set for the function tpconnect() or tpsend(), it is ignored.

(5) Event reception

If an event exists in the descriptor cd that identifies the connection, it can be received by the function (tpsend() or tprecv()) for communication based on the conversational service paradigm. The event provides communication-related information. For details, see the applicable *OpenTP1 Programming Reference* manual.

5.1.5 Notes on using xatmi interface for communication under OpenTP1

When using the XATMI interface for communication under OpenTP1, note the following:

- The following value must be specified in the user service definition of the user server which uses the XATMI interface.
  ```
  server_type = "xatmi"
  ```
- The XATMI interface does not have the concept of service groups. To use the XATMI interface for communication under OpenTP1, however, a service group

should be specified in the UAP user service definition.

- The following values must be specified in the user service definition or user service default definition when the XATMI interface is to be used under a transaction:
```
trn_expiration_time = non-zero value
trn_expiration_time_suspend = Y
```
- If blocking occurs when the function `tpcall()`, `tpacall()`, `tpconnect()`, or `tpsend()` sends data and is not removed within the specified period, `TPESYSTEM` is returned regardless of whether the `TPENOBLOCK` flag is set. The period until the function returns with a `TPESYSTEM` error is determined by the service request retry count and interval specified in the definition.
- If a transaction timeout occurs, the process terminates abnormally without returning `TPETIME`.
- When a UAP calls an XATMI interface function (e.g., `tpcall()`) after the UAP has been called by the function `dc_rpc_call()`, the UAP should be linked to a stub created by specifying client definitions both in the RPC interface definition and the XATMI interface definition. See the applicable *OpenTP1 Programming Reference* manual.
- If the transaction is settled by the function `tx_commit()` or other similar means, all data yet to be received becomes invalid.
- Before the internode load-balancing facility and the extended internode load-balancing facility can be used, as in the case of the function `dc_rpc_call()`, multiple SPP service group names must match in the user service definition. For this purpose, specify the service names and executable file names in the user service definition so that they match. If service names do not match, the function `tpcall()`, `tpacall()`, or `tpconnect()` may fail. If executable file names do not match, the result varies depending on which server UAP is scheduled.
- The maximum data length that can be transferred via the XATMI interface API is 500 Kbytes.

5.1.6 Communication data types

XATMI-interfaced communication allows structures in C or records in COBOL to be sent and received so that a chunk of data of some size can be transmitted with a single service request. Such a chunk of data is referred to as a *typed buffer* in C or a *typed record* in COBOL.

(1) Types and subtypes

Communication data belongs to a type and a subtype. The type and subtype of communication data to be used by a UAP are specified in the stub source file (XATMI

interface definition) which is used for creating the UAP. For XATMI interface definitions, see the applicable *OpenTP1 Programming Reference* manual.

(a) Type

A type identifies a kind of communication data as defined in the XATMI interface. Each type is characterized by the following:

- `X_OCTET`

  Data of this type is an octet array (byte string). The `X_OCTET` type has no subtype.

- `X_COMMON`

  Data of this type is a structure or record that is not nested. A subtype identifies the construction.

- `X_C_TYPE`

  Data of this type is a structure or record that is not nested. A subtype identifies the construction.

(b) Subtype

A subtype identifies a structure or record whose elements are populated with data in a range compatible with the type.

For the data types that can be used as communication data types, see (3) in *5.1.6 Communication data types*.

(2) How to use communication data types

The use of typed buffers or typed records allows structures in C or records in COBOL to be transferred. If function flags are specified appropriately, it is possible to receive data of a type or subtype of a different data type or of a different size from that specified for reception. However, before a communication data type can be handled by a UAP, it must match the value specified for the UAP in advance in the XATMI interface definition.

(3) List of data types that can be used with each communication data type

If you want to use a typed buffer, define its type, subtype, and data type in the XATMI interface definition (for the server UAP). When a stub is created from the XATMI interface definition file and the stub object file is linked to the server UAP, the typed buffer can be used. For XATMI interface definitions, see the applicable *OpenTP1 Programming Reference* manual.

Even when OSI TP is used as the protocol for communication with a non-OpenTP1 system, a typed buffer or type record can be sent to the remote system after it is converted so that it will be recognized by the remote system.

The table below lists the data types that can be used with each communication data type. The identifier indicates the data type that must be specified in the XATMI

interface definition. The data type in C and the data in COBOL indicates the typed buffer or typed record that is actually defined in the stub. To convert a data type as preparation for communication with a non-OpenTP1 system, specify the identifier requiring conversion in the XATMI interface definition.

Table 5-4: Data types that can be used with each communication data type

Type	Identifier	Data type in C	Data in COBOL	Communication protocol		Remarks
				TCP/IP	OSI TP	
X_OCTET	__#1	__#1	__#1	Y	Y	None
X_COMMON	short a	short a	PIC S9(9) COMP-5	Y	Y	None
	short a[*n*]	short a[*n*]	PIC S9(9) COMP-5 OCCURS *n* TIMES	Y	Y	None
	long a	long a	PIC S9(9) COMP-5	Y	Y	None
	long a[*n*]	long a[*n*]	PIC S9(9) COMP-5 OCCURS *n* TIMES	Y	Y	None
	char a#2	char a	PIC X	Y	Y	Array not to be converted
	octet a	char a	PIC X	Y	Y	Array not to be converted
	tchar a	char a	PIC X	-	Y	Array to be converted
	char a[*n*]#2	char a[*n*]	PIC X(*n*)	Y	Y	Array not to be converted
	octet a[*n*]	char a[*n*]	PIC X(*n*)	Y	Y	Array not to be converted
	tchar a[*n*]	char a[*n*]	PIC X(*n*)	-	Y	Array to be converted
X_C_TYPE	short a	short a	--	Y	N	None
	short a[*n*]	short a[*n*]	--	Y	N	None
	long a	DCLONG a	--	Y	N	None

Type	Identifier	Data type in C	Data in COBOL	Communication protocol		Remarks
				TCP/IP	OSI TP	
	long a[*n*]	DCLONG a[*n*]	--	Y	N	None
	int4 a	DCLONG a	--	Y	N	None
	int4 a[*n*]	DCLONG a[*n*]	--	Y	N	None
	char a[#2]	char a	--	Y	N	None
	octet a	char a	--	Y	N	None
	tchar a	char a	--	Y	N	None
	char a[*n*][#2]	char a[*n*]	--	Y	N	None
	octet a[*n*]	char a [*n*]	--	Y	N	None
	tchar a[*n*]	char a[*n*]	--	Y	N	None
	float a	float a	--	Y	N	None
	float a[*n*]	float a[*n*]	--	Y	N	None
	double a	double a	--	Y	N	None
	double a[*n*]	double a[*n*]	--	Y	N	None
	octet a[*n*][*n*]	char a[*n*][*n*]	--	Y	N	None
	tchar a[*n*][*n*]	char a[*n*][*n*]	--	Y	N	None
	str a[*n*]	char a[*n*]	--	Y	N	None
	str a[*n*][*n*]	char a[*n*][*n*]	--	Y	N	None
	tstr a[*n*]	char a[*n*]	--	Y	N	None
	tstr a[*n*][*n*]	char a[*n*][*n*]	--	Y	N	None

Legend:

Y: Can be used under this communication protocol.

N: Cannot be used under this communication protocol.

--: Always treated as an identifier not to be converted.

#1

`X_OCTET` will always be recognized, regardless of whether it is defined. If `X_OCTET` is specified in the `XATMI` interface definition, the execution of a stub creation command will encounter an error.

#2

This identifier can be used, but the following identifier should be used if you are in process of creation.

octet or tchar for `X_COMMON`

str or tstr for `X_C_TYPE`

(4) How to use functions which manipulate typed buffers

Explained below is how to use XATMI interface functions for manipulating communication data. The API that can manipulate communication data can be used only via the C language. There is no COBOL API for manipulating communication data.

(a) Allocation of typed buffer

To allocate a typed buffer, issue the function `tpalloc()` with a type and a subtype values from the UAP. The area allocated by the function `tpalloc()` is cleared to NULL.

(b) Reallocation of typed buffer

To expand a typed buffer, use the function `tprealloc()`. The typed buffer available with the function `tprealloc()` is only `X_OCTET`. If another typed buffer is specified, the function returns with an error. If the new buffer length is shorter than the data, the data is truncated. If the new buffer length is longer than the data, the extra area is cleared to NULL.

If reallocation fails, the old typed buffer is also invalidated.

(c) Deallocation of typed buffer

To deallocate the allocated area, call the function `tpfree()` having a pointer to the typed buffer as its argument. A value which is not a typed buffer pointer is ignored even if it is specified.

(d) Acquisition of typed buffer information

To acquire the type or other information about a buffer, call the function `tptypes()`.

(e) Notes on typed buffer operation

Do not use functions for typed buffer operation in combination with the functions `malloc()`, `realloc()`, or `free()` in the C library. For example, the buffer allocated by the function `tpalloc()` cannot be deallocated by the function `free()`. If the function `free()` is called for the allocated typed buffer, the result is unpredictable.

(5) Notes on using the X_OCTET type

The typed buffer of the `X_OCTET` type is partially different from other typed buffers. Notes on using the typed buffer of the `X_OCTET` type are given below.

1. The typed buffer of the `X_OCTET` type has no `subtype` (structure) (no `subtype`-specific information is needed).
2. Data is send without conversion (data is treated only as a bit array).
3. A parameter indicating the length must be specified.

5.1.7 How to create server UAP

This subsection explains how to use a function from the service function (server UAP) for XATMI-interfaced communication. The creation method of server UAPs is different from that of the OpenTP1 service functions.

(1) Coding service functions provided by server UAP

When writing code in C for service functions, follow the format given for `tpservice()`. This format is a standard coding format.

If you are writing code in COBOL, invoke `TPSVCSTART` before using an XATMI-interfaced API for service program processing.

(a) How to terminate a service function

A service function terminates when the function `tpreturn()` [`TPRETURN`] is called. For XATMI-interfaced communication under OpenTP1, the function `tpreturn()` must be called immediately before the service function is terminated by `return()`. The result of any process performed after call of the function `tpreturn()` is unpredictable.

(b) Advertisement of a service name

The server UAP can declare that its own service name is ready to offer service (advertisement of a service name). To advertise a service name, call the function `tpadvertise()` [`TPADVERTISE`].

When the server UAP is activated, the service specified in the user service definition becomes advertised service. It is unnecessary to call the function `tpadvertise()` with the service name specified in the user service definition.

To unadvertise a service name, call the function `tpunadvertise()`

[TPUNADVERTISE]. When the function tpunadvertise() is called, the service request for the service name returns with an error. A service name that once unadvertised by the function tpunadvertise() can accept a service request again if the function tpadvertise() is called.

The functions tpadvertise() and tpunadvertise() can be called only from SPPs. They cannot be called before the function dc_rpc_mainloop().

The function tpadvertise() has the function address to be advertised as its argument. This argument is used to check whether the service name can be advertised. Under OpenTP1, if the server using the function tpadvertise() has the same service group as the server advertising its service name, it is regarded as advertised and the function returns normally. If the service groups are different, the function tpadvertise() returns with an error.

(2) Relationship between load-balancing types (on one node or multiple nodes) and tpunadvertise()

(a) Load balancing at only one node (multiserver facility)

If the function tpunadvertise() is called from a process when load balancing is effective at one node (multiserver facility), all processes on which load is distributed become unable to receive service. Then, the function tpadvertise() is called to advertise the service, the processes become ready to accept a service request.

The multiserver facility is available only for queue-receiving servers (SPPs scheduled by schedule service). Servers that receive requests from socket cannot use the multiserver facility.

(b) Load balancing at multiple nodes (internode load-balancing facility and extended internode load-balancing facility)

Suppose that load balancing (internode load-balancing facility and extended internode load-balancing facility) is effective at multiple nodes and the function tpunadvertise() is called from a process. At the node of the process that called the function tpunadvertise(), service stops. However, the service request can be accepted by a server at another node. Under this environment, the function tpadvertise() is called to advertise the service, the process becomes ready for accepting service request.

The internode load-balancing facility can be used by either the queue or server that receives requests from socket. The extended internode load-balancing facility can only be used for queue-receiving servers.

5.1.8 Relationship between OpenTP1 facility and XATMI interface

(1) UAP trace

The UAP trace is also obtained for XATMI-interfaced communication. For details on the UAP trace, see the *OpenTP1 Tester and UAP Trace User's Guide*.

(2) Statistical information

The obtained operation statistical information is added to RPC information. Under OpenTP1 of this version, however, a part of fault information (such as XATMI interface specific faults) is not acquired. In particular, since the conversational service is specific to XATMI-interfaced communication, statistical information about the functions `tpsend()` and `tprecv()` cannot be acquired.

(3) RPC trace

The RPC trace can also be acquired. Under OpenTP1 of this version, however, a part of fault information (such as XATMI interface specific faults) is not acquired. In particular, since the conversational service is specific to XATMI-interfaced communication, RPC traces about the functions `tpsend()` and `tprecv()` cannot be acquired.

(4) Online tester

When you test UAPs using the XATMI interface, you cannot use some online tester facilities. Table 5-5 indicates the relationship between online tester facilities and the XATMI interface.

Table 5-5: Relationship between online tester facilities and XATMI interface

Online tester facility	Communication protocol used on XATMI interface	
	TCP/IP	OSI TP
Client UAP simulation	Y	N
Server UAP simulation	Y	N
MCF simulation	--	--
Resource update annulling	Y	Y
Command simulation	--	--
Tester file creation and editing	Y	N
UAP trace information acquisition	Y	Y
UAP trace information merge and editing output	Y	Y
Send message editing	--	--
Debugger connecting	Y	N

Legend:

Y: Can be used under the communication protocol.

N: Cannot be used under the communication protocol.

--: Irrelevant to XATMI-interfaced communication

5.2 TX interface (transaction control)

5.2.1 TX interfaces usable with OpenTP1

The API (TX_ functions) complying with X/Open can be used with UAPs of OpenTP1. UAPs which perform transaction control with TX_ functions can use other vendors' RM having the specifications conforming to X/Open.

(1) Relationship between OpenTP1 UAPs and TX_ functions

Table 5-6 lists the TX_ functions that can be used with UAPs of OpenTP1. Table 5-7 shows the relationship between OpenTP1 UAPs and TX_ functions.

Table 5-6: TX_ functions available with OpenTP1 UAPs

TX_ function name		Facility
C language	COBOL language	
tx_begin	TXBEGIN	Begin a global transaction
tx_close	TXCLOSE	Close a set of resource managers
tx_commit	TXCOMMIT	Commit a global transaction Chained mode (TX_CHAINED specified) Unchained mode (TX_UNCHAINED specified)
tx_info	TXINFORM	Return global transaction information
tx_open	TXOPEN	Open a set of resource managers
tx_rollback	TXROLLBACK	Roll back a global transaction Chained mode (TX_CHAINED specified) Unchained mode (TX_UNCHAINED specified)
tx_set_commit_return	TXSETCOMMITRET	Set commit_return characteristics
tx_set_transaction_control	TXSETTRANCTL	Set transaction_control characteristics
tx_set_transaction_timeout	TXSETTIMEOUT	Set transaction_timeout characteristics

Table 5-7: Relationship between OpenTP1 UAPs and TX_ functions

TX_ function name	SUP		SPP			MHP		UAP that handles offline work
	Not within transaction processing range	Within transaction processing range (root)	Not within transaction processing range	Transaction range		Not within transaction processing range	Within transaction processing range (root)	
				Root	Not root			
tx_begin	Y	--	Y	--	--	--	--	--
tx_close	Y	--	Y	--	--	--	--	--
x_commit (TX_CHAINED specified)	--	Y	--	Y	--	--	--	--
tx_commit (TX_UNCHAINED specified)	--	Y	--	Y	--	--	--	--
tx_info	Y	Y	Y	Y	Y	--	--	--
tx_open	Y		Y		--	--	--	--
tx_rollback (TX_CHAINED specified)	--	Y	--	Y	--	--	--	--
tx_rollback (TX_UNCHAINED specified)	--	Y	--	Y	Y	--	--	--
tx_set_commit_return	Y	Y	Y	Y	Y	--	--	--
tx_set_transaction_control	Y	Y	Y	Y	Y	--	--	--
tx_set_transaction_timeout	Y	Y	Y	Y	Y	--	--	--

Legend:

Y: Can be used with relevant UAP.

--: Cannot be used with relevant UAP.

5.2.2 How to use TX_ functions

(1) Starting a transaction

To start a transaction by TX_ functions, call the functions from UAP as shown below. To call the functions in this order, the transaction can be started regardless of the

atomic_update specification in the user service definition. If atomic_update = Y is specified in the user service definition, transaction processing start can be called unless tx_open() [TXOPEN] and tx_close() [TXCLOSE] are called.

Starting a transaction by SUP

```
dc_rpc_open()
tx_open()
tx_begin()
      :
      :
tx_commit() (synchronization point processing)
tx_close()
      :     tx_open( ) and tx_close( ) can be reissued in this section.
dc_rpc_close()
```

Starting a transaction by SPP (starting by service functions)

```
dc_rpc_open()
tx_open()
dc_rpc_mainloop()
      :
(Service function processing)
tx_begin()
      :
      :
tx_commit() (synchronization point processing)
      :
(Main function dc_rpc_mainloop() returns.)
tx_close()
      :
dc_rpc_close()
```

When starting a transaction within the service functions of SPP, call tx_open() before the function dc_rpc_mainloop().

(2) Obtaining synchronization point

Transaction processing started by tx_begin() [TXBEGIN] must always be completed by the function that obtains synchronization point (tx_commit() [TXCOMMIT] or tx_rollback() [TXROLLBACK]).

The chained mode (TX_CHAINED) and unchained mode (TX_UNCHAINED) are provided for tx_commit() and tx_rollback(). When starting a new global transaction after obtaining synchronization point, set the chained mode. When terminating transaction processing without starting a new transaction, set the unchained mode. The chained or unchained mode is set as a transaction_control characteristic by tx_set_transaction_control() [TXSETTRANCTL].

(3) Setting transaction characteristics

Characteristics of transaction processing can be set by TX_ functions.

(a) commit_return characteristic

When a transaction is committed in two phases, you can set whether control returns upon completion of the first or second phase. With the corresponding version of OpenTP1, return before completion of the second phase cannot be set. If such return is set, control returns with error. `commit_return` characteristic is set by `tx_set_commit_return()` [`TXSETCOMMITRET`].

(b) transaction_control characteristic

Sets whether a new transaction is to be started (chained mode or unchained mode) after the synchronization point (`tx_commit()` or `tx_rollback()`). For `transaction_control` characteristic, set either `TX_CHAINED` or `TX_UNCHAINED` by `tx_set_transaction_control()` [`TXSETTRANCTL`].

(c) transaction_timeout characteristic

Monitoring time for transactions can be set. The `transaction_timeout` characteristic is set by `tx_set_transaction_timeout()` [`TXSETTIMEOUT`]. The `transaction_timeout` characteristic set by `tx_set_transaction_timeout()` has priority over the value of `trn_expiration_time` defined by the system.

(4) Obtaining transaction information

`tx_info()` [`TXINFORM`] enables the structures containing transaction branch IDs or transaction characteristics to be referenced.

The following shows the formats of the structures that can be referenced.

```
XID                    xid;
COMMIT_RETURN          when_return;
TRANSACTION_CONTROL    transaction_control;
TRANSACTION_TIMEOUT    transaction_timeout;
TRANSACTION_STATE      transaction_state;
```

(5) Relation to user service definition

By using `tx_open()` and `tx_close()`, processing after `tx_begin()` can be processed as a transaction, regardless of the value of `atomic_update` in the user service definition.

When a service is called by the UAP which called `tx_begin()`, processing of the service is included in the global transaction, regardless of the specification of `atomic_update` in the server UAP.

5.2.3 Restrictions on using TX_ functions

(1) Use of OpenTP1 functions when using TX_ functions

OpenTP1 functions can be used along with TX_ functions. However, do not use `TX_` functions together with the transaction control functions (`dc_trn_` ~) of OpenTP1. Transaction control cannot be performed with mixed use of both facilities.

(2) Relationship between dc_rpc_open() and tx_open()

Call the function dc_rpc_open() before tx_open(). If tx_open() is called without the function dc_rpc_open(), error is returned with TX_ERROR.

(3) Difference between dc_rpc_close() and tx_close()

The function dc_rpc_open() cannot be called after the function dc_rpc_close() is called. However, tx_open() can be called after tx_close() is called. When the UAP is to be placed in dormant state due to traffic, call tx_close(), then recall tx_open().

(4) Relationship between tx_open()/tx_close() and open/close functions specific to RM

tx_open() and tx_close() are the functions which inform each RM that access was requested by UAP or it ended. By using tx_open() or tx_close(), processing requests from UAP are posted to each RM.

The open and close functions specific to RM (e.g., dc_dam_open(), dc_dam_close()) indicate the start and end of actual processing. If tx_open() or tx_close() is called, the open and close functions specific to RM do not become unnecessary.

5.2.4 Comparison with transaction control functions of OpenTP1 (dc_trn_ ~)

(1) Correspondence between TX_ functions and transaction control functions of OpenTP1 (dc_trn_ ~)

The table below shows the relationship between TX_ functions and transaction control functions of OpenTP1 (dc_trn_ ~).

Table 5-8: Relationship between TX_ functions and transaction control functions of OpenTP1 (dc_trn_ ~)

TX_ function name	OpenTP1 transaction control function (dc_trn_ ~)
tx_begin()	dc_trn_begin()
tx_close()	No corresponding function
tx_commit() (TX_CHAINED specified)	dc_trn_chained_commit()
tx_commit() (TX_UNCHAINED specified)	dc_trn_unchained_commit()
tx_info()	dc_trn_info()
tx_open()	No corresponding function

TX_ function name	OpenTP1 transaction control function (dc_trn_ ~)
tx_rollback() (TX_CHAINED specified)	dc_trn_chained_rollback()
tx_rollback() (TX_UNCHAINED specified)	dc_trn_unchained_rollback()
tx_set_commit_return()	No corresponding function
tx_set_transaction_control()	No corresponding function
tx_set_transaction_timeout()	No corresponding function

(2) Time monitoring with TX_ function

Elapsed time of a transaction can be monitored with tx_set_transaction_timeout(). In this case, the transaction_timeout characteristic set by tx_set_transaction_timeout() has priority over the value of trn_expiration_time defined by the system.

(a) Range of time monitoring

For time monitoring from tx_begin() to the synchronization point (tx_commit() or tx_rollback()), the following can be selected:

Whether or not the time until the function dc_rpc_call() called in the transaction returns is to be included. The range of transaction monitoring time can be specified by trn_expiration_time_suspend in the user service definition, user service default definition, or transaction service definition. For details on the value to be assigned to trn_expiration_time_suspend and transaction time monitoring, see the manual *OpenTP1 System Definition*.

Chapter

6. X/Open-compliant Inter-application Communication (TxRPC)

This chapter explains what facilities are available when the X/Open-compliant inter-application communication method (TxRPC interface) is used with OpenTP1 application programs.

This chapter contains the following sections:

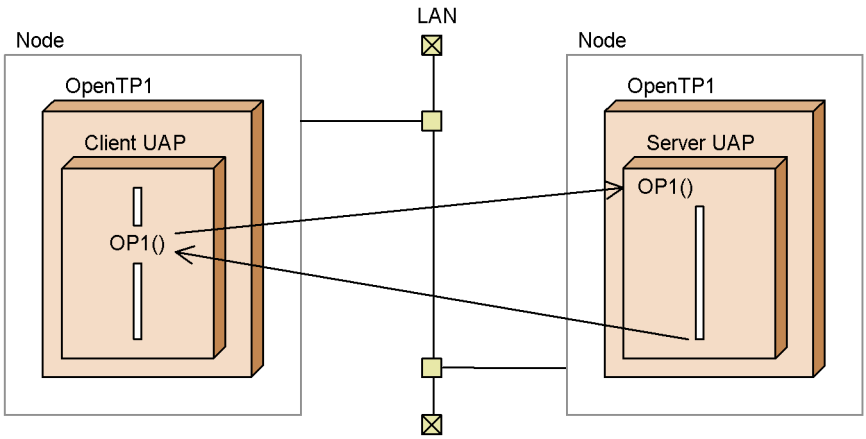
6.1 Communication through TxRPC interface
6.2 Communication allowed with application programs
6.3 Procedures for creating application programs for TxRPC communication

6.1 Communication through TxRPC interface

The TxRPC interface provides an X/Open-compliant client/server-mode communication method. The TxRPC interface can be used for communication between UAP processes under OpenTP1. Unlike other types of client/server-mode communication, TxRPC communication is performed by calling a user-created function from the client. The user can create a function without consideration of the lower layers of communication protocol.

The figure below shows the outline of communication through the TxRPC interface.

Figure 6-1: Outline of communication through TxRPC interface



6.1.1 Types of TxRPC communication

The TxRPC communication method is classified into the following two types depending on whether DCE RPC is used:

- IDL-only TxRPC
- RPC TxRPC

(1) IDL-only TxRPC

With this method, a UAP is created using only files generated by the IDL compiler. DCE need not be set up with this method.

(2) RPC TxRPC

With this method, a DCE RPC is used for the communication protocol. This version does not support RPC TxRPC.

6.1.2 Application programs that can be created

The following application programs can be created as UAPs used for TxRPC communication:

- Client UAP (SUP)
- Server UAP (SPP)

(1) Process type

One of the following options is given to the `txidl` command to specify the type of a UAP process to be created. This is called a *process type*.

- ndce:

  Indicates a process which does not use DCE. This option is specified for a SUP or a SPP.

For the syntax of the `txidl` command, see the manual *OpenTP1 Programming Reference C Language*.

6.1.3 Necessary libraries

The prerequisite library for TxRPC communication is as follows.

When SUP or SPP is created

The following product must be installed in the system:

- TP1/Server Base

6.2 Communication allowed with application programs

TxRPC allows the following types of communication:

- Synchronous-response-type transactional RPC
- Synchronous-response-type nontransactional RPC

6.2.1 TxRPC remote procedure calls

With a TxRPC remote procedure call (RPC), a user-created function is called. The only available RPC mode is synchronous-response-type RPC. The other RPC modes (asynchronous-response-type RPC, nonresponse-type RPC, and conversational RPCs) are unavailable.

For TxRPC communication, a function to be called is called a *manager*.

The OpenTP1 RPC (the function `dc_rpc_call()`) can also be used with any type of TxRPC communication.

6.2.2 TxRPC transaction processing

A TxRPC UAP allows transaction processing. To control transactions by a UAP process, use a `TX_` function (e.g., `tx_begin()` or `tx_rollback()`). For details on transaction control using the `TX_` functions, see *5.2 TX interface (transaction control)*.

(1) Scope of transaction processing

With OpenTP1 TxRPC, transaction processing is permitted for IDL-only TxRPC.

(2) Transaction attributes

A transaction attribute must be specified for a TxRPC UAP process which performs transaction processing. TxRPC has the following transaction attributes:

- `transaction_mandatory`

  This attribute always indicates transaction extension. If a UAP with this attribute is called from a non-transaction process, an error occurs.

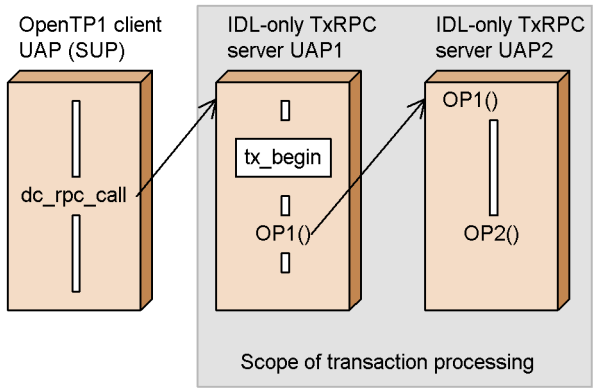
- `transaction_optional`

  When a UAP with this attribute is called from a transaction process, a UAP process is treated as a transaction. When a UAP with this attribute is called from a non-transaction process, a UAP process is treated as a non-transaction process.

A transaction attribute of a UAP is specified in the interface definition file (IDL file). Either `transaction_mandatory` or `transaction_optional` may be specified.

The figure below shows communication by using application programs.

Figure 6-2: Communication by using application programs



6.2.3 Relation between application programs using OpenTP1 facilities and TxRPC application programs

Communication between an SUP/SPP which uses the OpenTP1 remote procedure call (the function `dc_rpc_call()`) and a server UAP which uses TxRPC is performed as follows:

- An SPP called using the function `dc_rpc_call()` can call another SPP using TxRPC.
- A TxRPC server UAP can request an SPP for service using the function `dc_rpc_call()`. However, any server UAP with process type `nbet` cannot use this function.

6.3 Procedures for creating application programs for TxRPC communication

This section explains the procedures for creating UAPs which use the TxRPC communication methods.

6.3.1 Procedure for creating UAP for IDL-only TxRPC communication

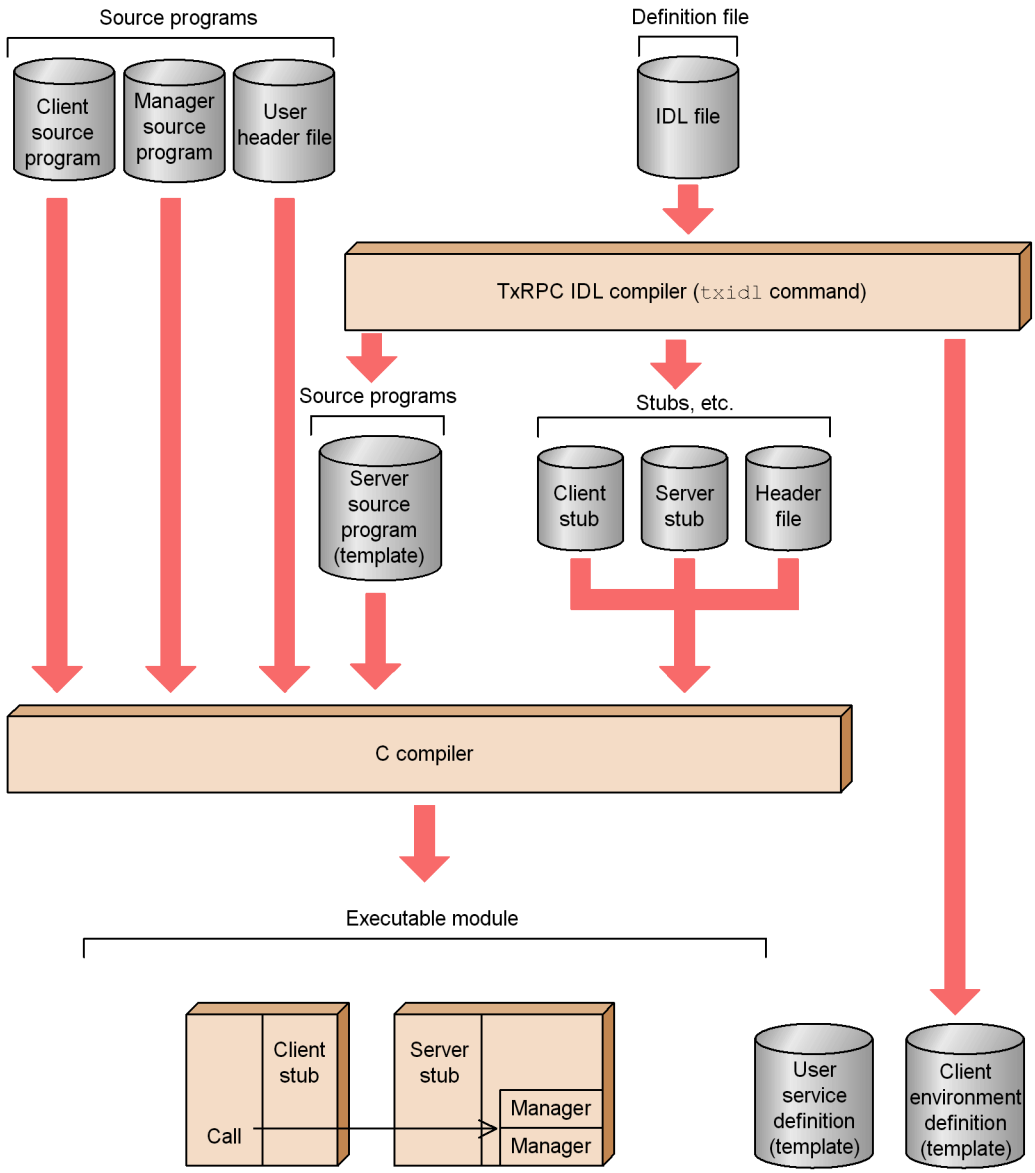
To create a UAP for IDL-only TxRPC communication:

1. Create an Interface Definition Language (IDL) file.
2. Compile the IDL file using the IDL compiler (`txidl` command).
3. Based on the template of a server UAP generated using the `txidl` command, code necessary programs along with the client UAP.
4. Use the C compiler to compile and link stubs generated using the `txidl` command and the coded programs.

For the procedure for creating a UAP for TxRPC communication, see the manual *OpenTP1 Programming Reference C Language*.

The figure below shows the procedure for creating a UAP for IDL-only TxRPC communication.

Figure 6-3: Procedure for creating UAP for IDL-only TxRPC communication



Chapter
7. Facilities Provided by TP1/Multi

This chapter explains the facilities available with TP1/Multi in cluster/parallel mode.

This chapter contains the following sections:

7.1 Application programs in cluster/parallel mode
7.2 Facilities available with the use of application programs
7.3 Conditions for using multinode facility functions

7.1 Application programs in cluster/parallel mode

This section explains the UAPs for OpenTP1 installed in the cluster system or parallel processing system.

7.1.1 Node on which application programs can be executed

In an environment using the multinode facility, UAPs (user servers) can be used on only archived journal nodes. They cannot be used on archiving journal nodes.

7.1.2 Prerequisites to application program execution

The following prerequisites should be satisfied by OpenTP1 nodes containing UAPs which execute the facilities for cluster system or parallel processing system:

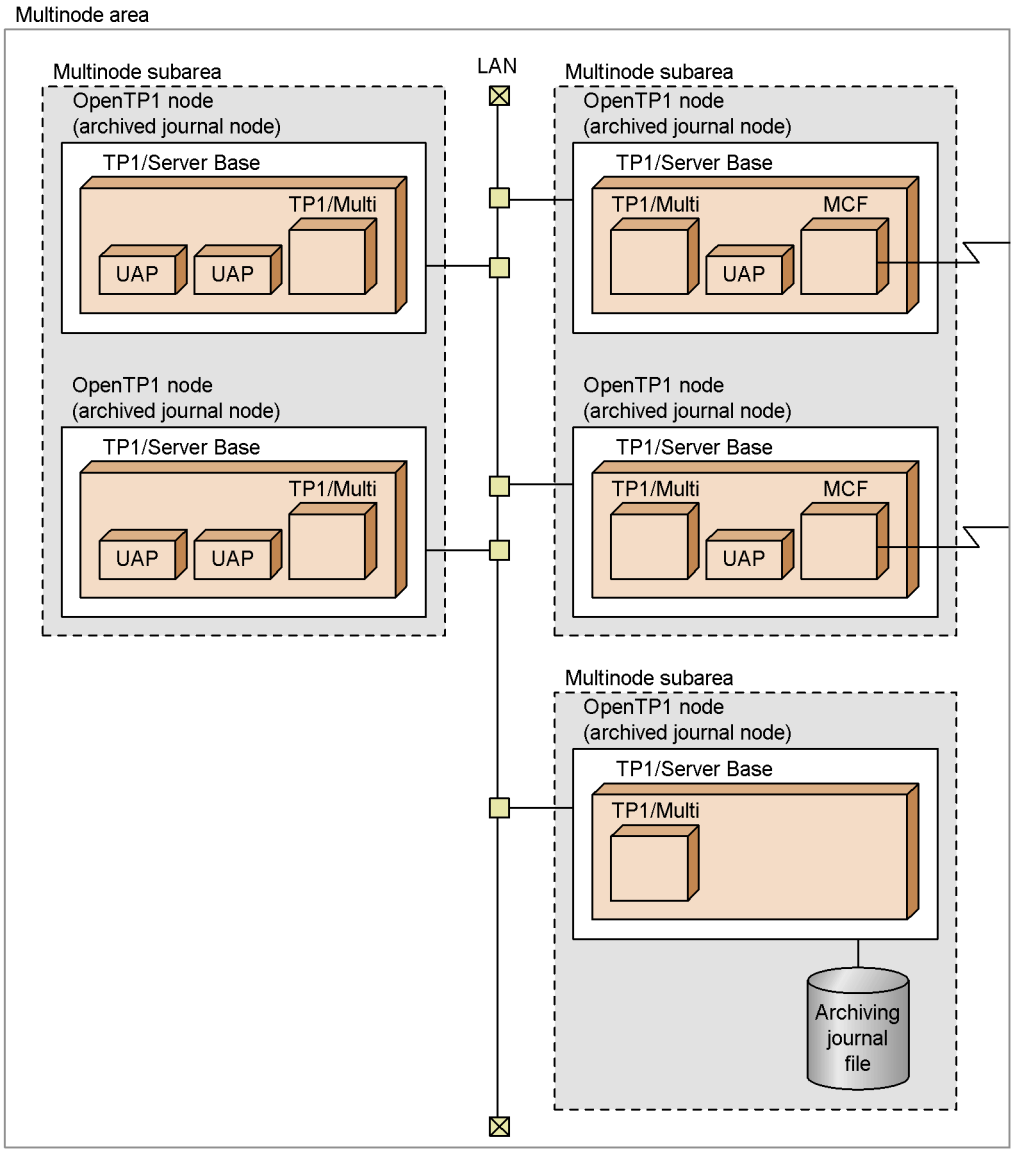
- TP1/Multi must be installed.
- `Y` must be assigned to `multi_node_option in` the system common definition.

However, none of the above prerequisites must be satisfied when:

- The user server status acquisition function (the function `dc_adm_get_sv_status()`) is used to acquire the status of the user server at the own node.
- The node identifier of the own OpenTP1 node acquisition function (the function `dc_adm_get_node_id()`) is used.

The figure below shows the outline of the application programs in cluster/parallel mode.

Figure 7-1: Outline of application programs in cluster/parallel mode



7.2 Facilities available with the use of application programs

This section explains the facilities which can be used by calling a function from the UAP in a cluster/parallel mode OpenTP1 system.

7.2.1 Acquisition of OpenTP1 node status

By using a function from a UAP, the status of an OpenTP1 node contained in a cluster/parallel mode system can be acquired. Through the acquisition of OpenTP1 node statuses, the multinode area or subarea can be monitored by the UAP.

The statuses that can be acquired are:

- The OpenTP1 node has not started.
- The OpenTP1 node is halted or is being terminated abnormally.
- The OpenTP1 node is normally being started.
- The OpenTP1 node is normally being restarted.
- The OpenTP1 node is online.
- The OpenTP1 node is normally being terminated.
- The OpenTP1 node is being terminated according to plan A.
- The OpenTP1 node is being terminated according to plan B.

Whether to acquire the statuses of multiple OpenTP1 nodes in succession or to acquire the status of only the specified OpenTP1 node can be specified.

(1) How to acquire the statuses of OpenTP1 nodes in succession

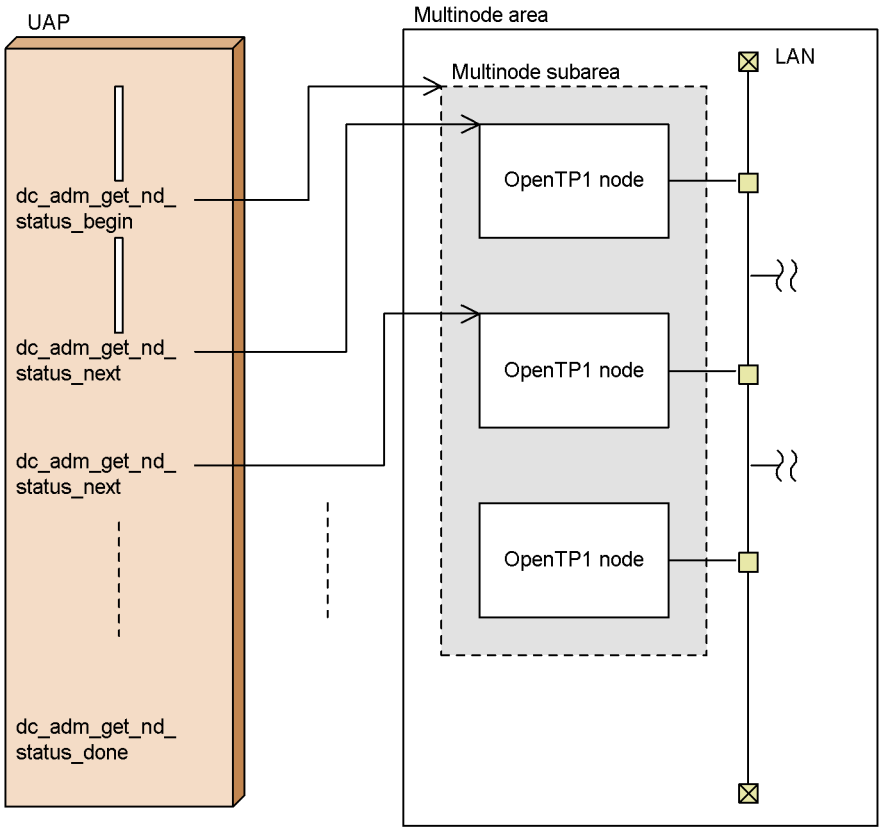
The statuses of all OpenTP1 nodes in each multinode area or subarea are acquired.

To acquire the statuses of all nodes in succession, call the function `dc_adm_get_nd_status_begin()` with the multinode area or subarea identifier specified. This function returns the number of OpenTP1 nodes in the specified area. Then, call the function `dc_adm_get_nd_status_next()` to acquire OpenTP1 node statuses. Continue calling the function `dc_adm_get_nd_status_next()` until the statuses of the last candidate node is acquired. Finally, call the function `dc_adm_get_nd_status_done()` to terminate status acquisition.

Once the function `dc_adm_get_nd_status_begin()` is called, it cannot be called (that is, the function `dc_adm_get_nd_status_begin()` cannot be nested).

The figure below shows the procedure for acquiring OpenTP1 node statuses in succession.

Figure 7-2: Procedure for acquiring OpenTP1 nodes in succession



(2) How to acquire the status of only the specified OpenTP1 node

To acquire the status of only the specified OpenTP1 node, call the function `dc_adm_get_nd_status()`. This function returns the status concerning the node identifier of the OpenTP1 node specified in the function.

7.2.2 Acquisition of user server status

By calling a function from a UAP, the statuses of user servers at OpenTP1 nodes making up a cluster/parallel mode system can be acquired. Through the acquisition of user server statuses, the user servers in the multinode subarea can be monitored.

The statuses that can be acquired are:

- The user server is halted or is being terminated abnormally.
- The user server is normally being started.
- The user server is normally being restarted.

- The user server is online.
- The user server is normally being terminated.

Whether to acquire the statuses of multiple user servers in succession or to acquire the status of only the specified user server can be specified.

(1) How to acquire the statuses of user servers in succession

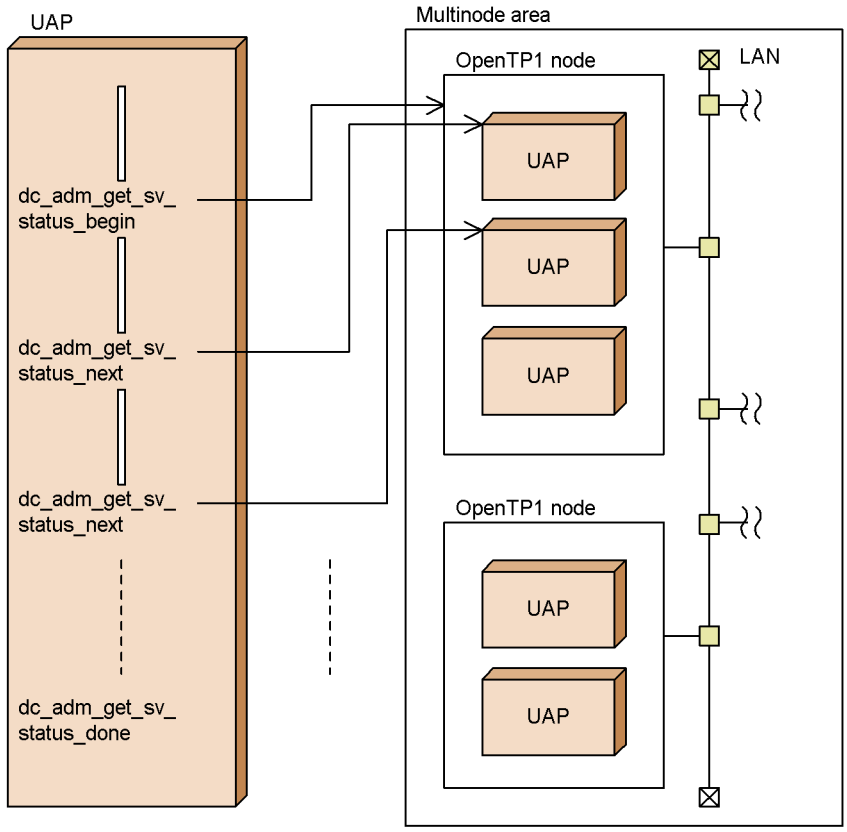
The statuses of all user servers at the node identified by the OpenTP1 node identifier are acquired.

To acquire the statuses of all user servers in succession, call the function `dc_adm_get_sv_status_begin()` with the node identifier specified. This function returns the number of user servers at the specified OpenTP1 node. Then, call the function `dc_adm_get_sv_status_next()` to acquire the user server statuses. Continue issuing the function `dc_adm_get_sv_status_next()` until the status of the last candidate user server is acquired. Finally, call the function `dc_adm_get_sv_status_done()` to terminate status acquisition.

Once the function `dc_adm_get_sv_status_begin()` is called, it cannot be called (that is, the function `dc_adm_get_sv_status_begin()` cannot be nested).

The figure below shows the procedure for acquiring user server statuses in succession.

Figure 7-3: Procedure for acquiring user server statuses in succession



(2) How to acquire the status of only the specified user server

To acquire the status of only the specified user server, call the function `dc_adm_get_sv_status()`. This function returns the status of the user server identified by the node identifier specified in the function.

7.2.3 Acquisition of OpenTP1 node identifier

By calling a function from a UAP, all node identifiers in the multinode area or subarea can be acquired. Through the acquisition of node identifiers, the UAP can recognizes which OpenTP1 nodes are contained in the multinode area or subarea.

Whether to acquire all node identifiers in succession or to acquire the node identifier of only the specified OpenTP1 node can be specified.

(1) How to acquire node identifiers of all OpenTP1 nodes in succession

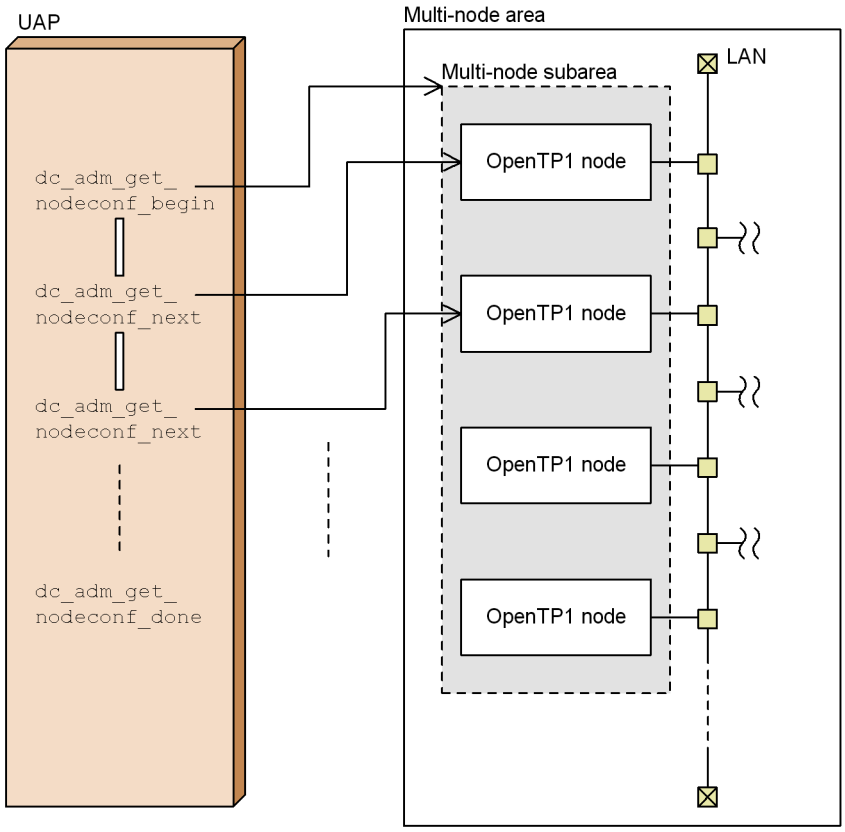
The node identifiers of all OpenTP1 nodes in each multinode area or subarea are

acquired. To acquire node identifiers in succession, call the function `dc_adm_get_nodeconf_begin()` with the area specified. This function returns the number of OpenTP1 nodes in the specified area. Then, call the function `dc_adm_get_nodeconf_next()` to acquire the OpenTP1 node identifiers. Continue using the function `dc_adm_get_nodeconf_next()` until the node identifier of the last candidate OpenTP1 node is acquired. Finally, issue the function `dc_adm_get_nodeconf_done()` to terminate status acquisition.

Once the function `dc_adm_get_nodeconf_begin()` is called, it cannot be called (that is, the function `dc_adm_get_nodeconf_begin()` cannot be nested).

The figure below shows the procedure for acquiring OpenTP1 node identifiers in succession.

Figure 7-4: Procedure for acquiring OpenTP1 node identifiers in succession



(2) How to acquire the node identifier of only the local OpenTP1 node

To acquire the node identifier of only the OpenTP1 node running the UAP, call the function `dc_adm_get_node_id()`. This function returns the node identifier of the

local OpenTP1 node.

7.3 Conditions for using multinode facility functions

The table below lists the conditions for using multinode facility functions.

Table 7-1: Conditions for using multinode facility functions

Multinode function name and specification of argument node_id		TP1/Multi installed		TP1/Multi not installed	
		Specification of multi_node_option		Specification of multi_node_option	
		Y	N	Y	N
dc_adm_get_nd_status_begin	Any	Y	N	--	N
dc_adm_get_nd_status_next	Any	Y	N	--	N
dc_adm_get_nd_status_done	Any	Y	N	--	N
dc_adm_get_nd_status	'*' specified	Y	N	--	N
	Local node_id specified	Y	N	--	N
	Another node_id specified	Y	N	--	N
dc_adm_get_sv_status_begin	'*' specified	Y	Y	--	Y
	Local node_id specified	Y	Y	--	Y
	Another node_id specified	Y	N	--	N
dc_adm_get_sv_status_next	'*' specified	Y	Y	--	Y
	Local node_id specified	Y	Y	--	Y
	Another node_id specified	Y	N	--	N

Multinode function name and specification of argument node_id		TP1/Multi installed		TP1/Multi not installed	
		Specification of multi_node_option		Specification of multi_node_option	
		Y	N	Y	N
dc_adm_get_sv_status_done	*' specified	Y	Y	--	Y
	Local node_id specified	Y	Y	--	Y
	Another node_id specified	Y	N	--	N
dc_adm_get_sv_status	*' specified	Y	Y	--	Y
	Local node_id specified	Y	Y	--	Y
	Another node_id specified	Y	N	--	N
dc_adm_get_nodeconf_begin	Any	Y	N	--	N
dc_adm_get_nodeconf_next	Any	Y	N	--	N
dc_adm_get_nodeconf_done	Any	Y	N	--	N
dc_adm_get_node_id	Any	Y	N	--	N

Legend:

Y: The function can be used under this condition.

N: The function cannot be used under this condition.

--: If the function is called under this condition, OpenTP1 terminates abnormally.

Chapter
8. OpenTP1 Samples

This chapter explains how to use samples given by OpenTP1.

This chapter contains the following sections:

8.1 Outline of samples
8.2 How to use Base sample
8.3 How to use DAM sample
8.4 How to use TAM sample
8.5 Specifications of sample programs
8.6 How to use MCF sample
8.7 Samples to be used to dispatch multi OpenTP1 command
8.8 COBOL language templates
8.9 How to use sample scenario template
8.10 How to use real-time acquisition item definition templates

8.1 Outline of samples

OpenTP1 provides samples to support the system setup. Using samples gives the following advantages:

- Workload from installation to setup of OpenTP1 is reduced.
- Tools for supporting system operation can be used.
- Templates can be used when a UAP is written in COBOL.

8.1.1 Types of sample programs

OpenTP1 samples are as follows:

- Base sample

  Comes with TP1/Server Base.

- DAM sample

  Comes with TP1/FS/Direct Access.

- TAM sample

  Comes with TP1/FS/Table Access.

- MCF sample

  Comes with the message control facility (TP1/Message Control, TP1/NET/Library, and communication protocol supporting product).

- `delvcmd` command

  Dispatches commands for MultiOpenTP1.

- COBOL language templates

  Are DATA DIVISION templates used when a UAP is written in COBOL.

- Sample scenario templates

  Is the sample scenario template used when the system is operated using a scenario template. To use this template, you must have a JP1 product (JP1/AJS2, JP1/AJS2 - Scenario Operation, or JP1/Base) that is a prerequisite for systems that use a scenario template.

- Real-time acquisition item definition templates

  Templates used for the real-time statistical information service.

These samples are independent of each other and stored in separate directories. They can be used by themselves.

(1) Files accessed from application programs

Base sample, DAM sample, and TAM sample can use sample programs (UAPs) and the OpenTP1 file system. UAPs in each sample use programs of the same format. However, the location for storing databases used by the UAPs varies depending on the sample as follows:

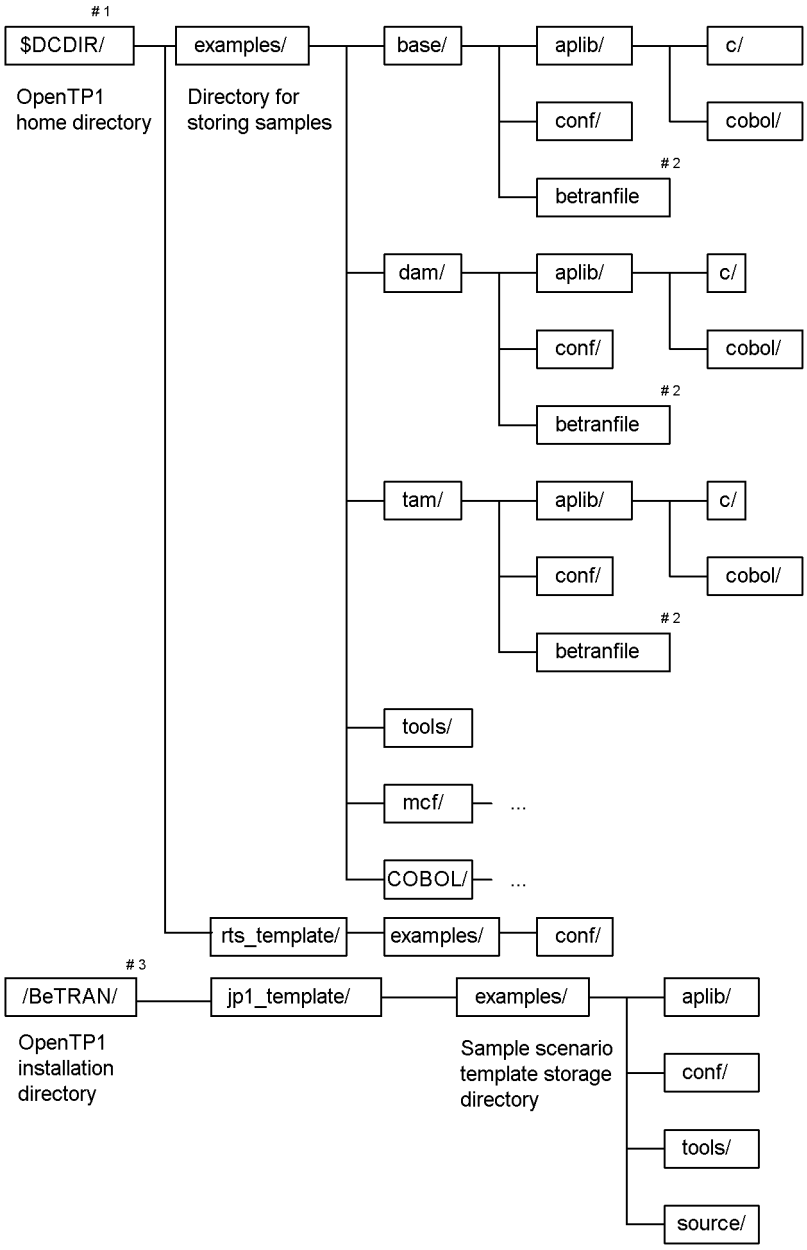
- Database for Base sample: On memory table
- Database for DAM sample: DAM file
- Database for TAM sample: TAM table

UAPs in each sample reference or update data in the database. The access procedure helps the user understand how to use the OpenTP1 API.

8.1.2 Sample program directory configuration

This section explains the files that are used with OpenTP1 samples. The figure below shows the directories containing OpenTP1 samples. `$DCDIR` is the OpenTP1 home directory.

Figure 8-1: Configuration of directories for storing samples



#1

`$DCDIR` is an environment variable indicating the OpenTP1 home directory. If

you have copied the OpenTP1 samples to a directory other than the default OpenTP1 home directory, `$DCDIR` is the name of that directory. If not, the `examples/` directory is placed under `$DCDIR`.

#2

`betranfile` is created when a sample command for tools is executed. `betranfile` does not exist at initial time.

#3

The directory for installation of OpenTP1 depends on the OS.

(1) Contents of examples/ directory under $DCDIR

The `examples/` directory under `$DCDIR` contains the directory used with the sample. Files in the `examples/` directory are listed below with a brief explanation.

`base/`

Directory that contains files for the Base sample

`dam/`

Directory that contains files for the DAM sample

`tam/`

Directory that contains files for the TAM sample

`tools/`

Directory that contains tools commonly used by all samples (tool directory)

`mcf/`

Directory that contains files for the message control facility (MCF) sample

`COBOL/`

Directory that contains COBOL language templates

(a) Contents of base/, dam/, and tam/ directories

Directories and files in the `base/`, `dam/`, and `tam/` directories are listed below with a brief explanation.

`aplib/`

Directory that contains sample UAPs

`c/`

Directory that contains source files (in the C language) of UAPs in the sample

`cobol/`

Directory that contains source files (in the COBOL language) of UAPs in the sample

`conf/`

Directory that contains definition files of the sample

`betranfile`

OpenTP1 file system for the sample (the contents of this file are created by the tools in the `tools/` directory)

(b) Contents of tools/ directory

Files in the `tools/` directory are listed below with a brief explanation.

`base_mkfs`

Shell file that creates an OpenTP1 file system for the Base sample

`dam_mkfs`

Shell file that creates an OpenTP1 file system for the DAM sample

`apbat`

File that creates a DAM file for the DAM sample (used by `dam_mkfs`). This file is created when a UAP executable file is created using the `make` command. For details on how to create an executable file, see *8.3.2(1)(b) Create a UAP executable file*.

`tam_mkfs`

Shell file that creates an OpenTP1 file system for the TAM sample

`tamdata`

Data file used to create a TAM table for `dam_mkfs`

`chconf`

Command that modifies a definition file (used to change `$DCDIR` to the actual OpenTP1 home directory)

`bkconf`

Command that restores a definition file modified by `chconf` to its original state

`delvcmd`

Command that delivers a command to nodes in Multi-OpenTP1 mode. For details on the `delvcmd` command, see *8.7 Samples to be used to dispatch multi OpenTP1 command*.

(c) Contents of mcf/ directory

For details on the configuration of the `mcf/` directory, see *8.6 How to use MCF sample*.

(d) Contents of COBOL/ directory

For details on the configuration of the `COBOL/` directory, see *8.8 COBOL language templates*.

(2) Contents of rts_template/ directory under $DCDIR

The `rts_template/` directory under `$DCDIR` directory contains the directory used with the real-time statistical information service. Files in the `rts_template/` directory are listed below with a brief explanation.

(a) Contents of examples/ directory

Directories in the `examples/` directory are listed below with a brief explanation.

`conf/`

Directory that contains the real-time acquisition item definition files for the real-time statistical information service

- `base_itm`

  Real-time acquisition item definition file for BASE

- `dam_itm`

  Real-time acquisition item definition file for DAM

- `tam_itm`

  Real-time acquisition item definition file for TAM

- `all_itm`

  Real-time acquisition item definition file for all statistical information

- `none_itm`

  Real-time acquisition item definition file for no statistical information

- `mcfs_itm`

  Real-time acquisition item definition file for the MCF (acquired for the entire system or for each server or service)

- `mcfl_itm`

  Real-time acquisition item definition file for the MCF (acquired for each logical terminal)

- `mcfg_itm`

  Real-time acquisition item definition file for the MCF (acquired for each

service group)

(3) Contents of jp1_template/ directory under the installation directory

The `jp1_template/` directory under the installation directory contains the directory used with the sample scenario template. Files in the `jp1_template/` directory are listed below with a brief explanation.

`examples/`

Directory that contains the files used with the scale-out sample scenario template

(a) Contents of examples/ directory

Directories in the `examples/` directory are listed below with a brief explanation.

`aplib/`

Directory that contains the load module (load module for the source file under the `source/` directory) for the sample program of the scenario template

`conf/`

Directory that contains the definition file for the sample scenario template

`tools/`

Directory that contains the shell file used for the sample scenario template

`dcjset_conf`

Shell file that sets the OpenTP1 environment for the sample scenario template

`dcj_mkfs`

Shell file that creates an OpenTP1 file system for the sample scenario template

`dcjmk_dcdir`

Shell file that creates an OpenTP1 directory for the sample scenario template

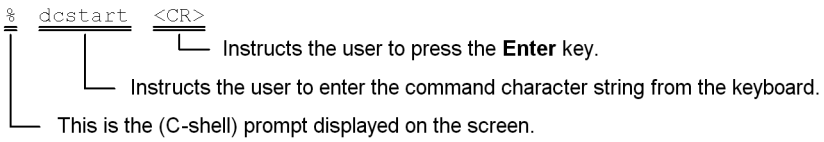
`source/`

Directory that contains the sample program (UAP) of the scenario template. The sample scenario template uses the Base sample as the sample program. For details on the specifications of the Base sample, see *8.5 Specifications of sample programs*.

8.1.3 Explanation format of samples

This section explains how to use OpenTP1 samples.

In this section, a command input example is written in the following format:



Commands that should be typed by the user are underlined.

The above example means that "after confirming the prompt displayed on the screen, type the command (`dcstart`) and press the Return key".

8.2 How to use Base sample

This section explains how to use the Base sample. The figure below shows the procedure for using samples.

Figure 8-2: Outline of procedure for using samples (Base sample when using a stub)

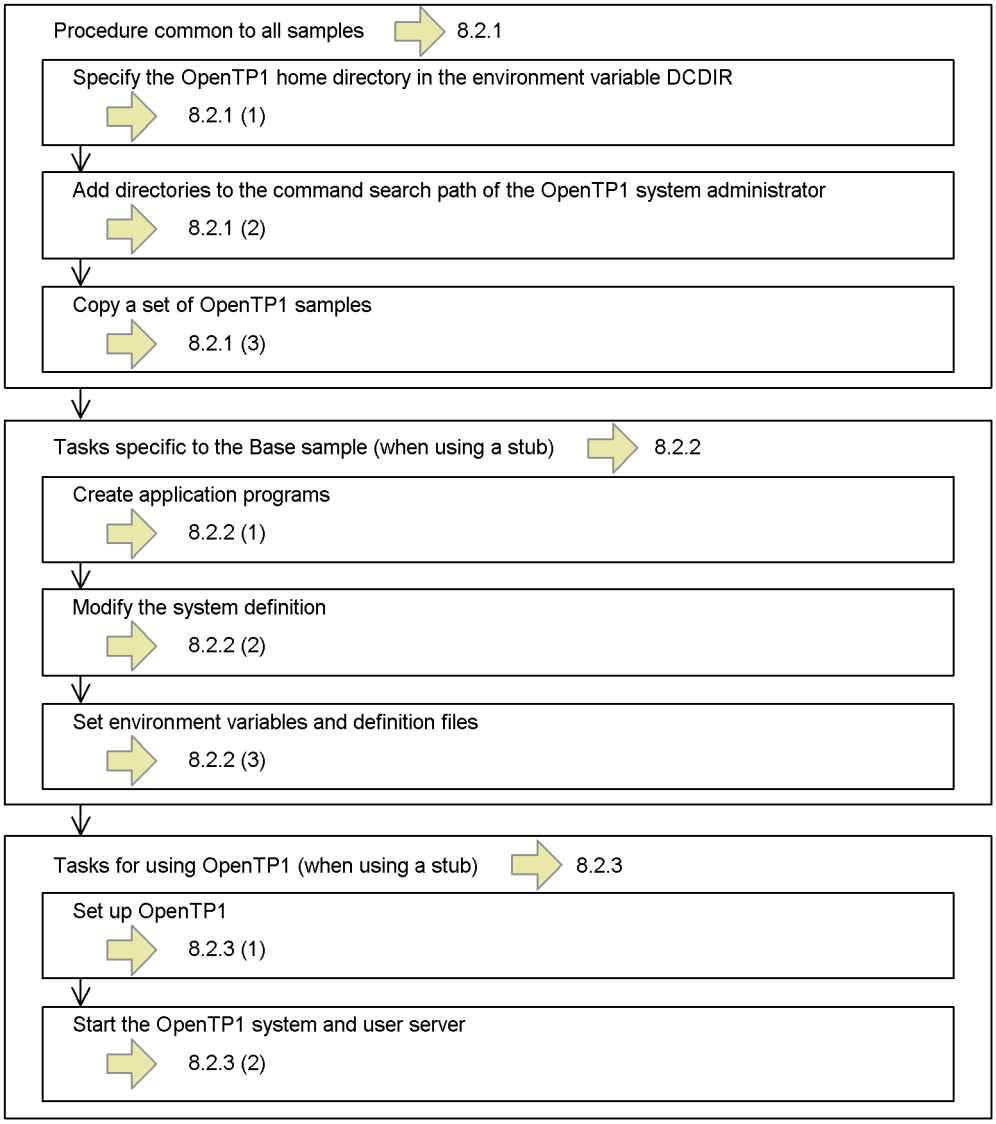
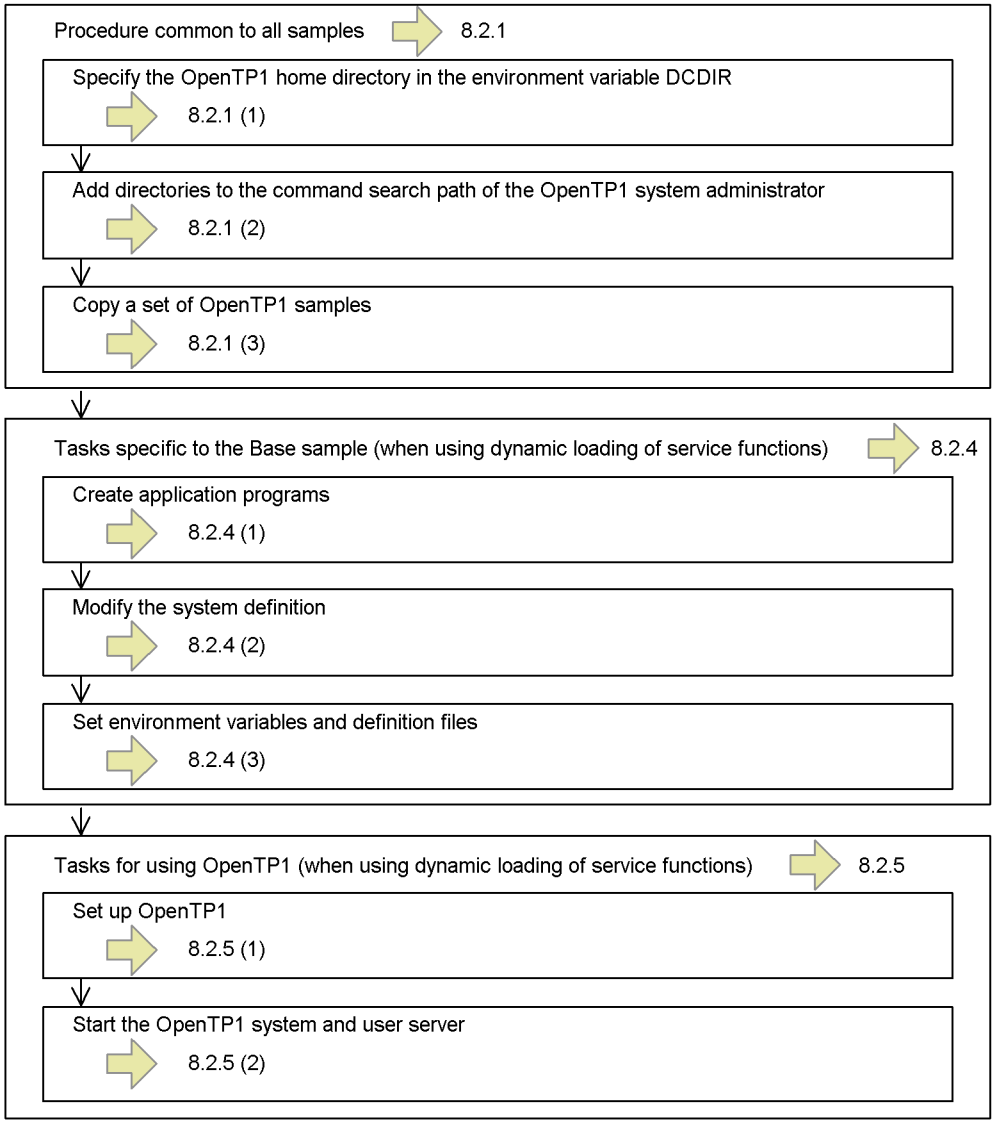


Figure 8-3: Outline of procedure for using samples (Base sample when using dynamic loading of service functions)



8.2.1 Procedure common to all samples (Base sample)

This subsection explains the preparation procedure common to three OpenTP1 samples (Base, DAM, and TAM samples).

Install OpenTP1 products (TP1/Server Base, TP1/FS/Direct Access, and TP1/FS/

Table Access) before using these samples. The OpenTP1 administrator is responsible for this work. The procedure so far is the same as the ordinary OpenTP1 setup procedure.

(1) Specifying an OpenTP1 home directory in environment variable DCDIR

Specify an OpenTP1 home directory in the environment variable `DCDIR`.

Example

Change the OpenTP1 home directory to `/usr/betran`:

```
% setenv DCDIR /usr/betran <CR>
```

(2) Adding directories to the OpenTP1 administrator's command search path

Add a search path for OpenTP1 commands and a search path for sample-serving tools to be used with samples to the OpenTP1 administrator's command search path.

- `$DCDIR/bin`: Search path for OpenTP1 commands
- `$DCDIR/examples/tools`: Search path for sample-serving tools

(3) Copying a set of OpenTP1 samples

If the OpenTP1 home directory was changed to a directory other than `/BeTRAN` in step (1), copy a set of OpenTP1 samples into the environment of the OpenTP1 home directory. If the OpenTP1 home directory is unchanged, copying is not needed. For copying, use the cp, tar, or other similar command.

Before copying, make sure that the OpenTP1 home directory is configured as shown in *8.1.2 Sample program directory configuration*. Otherwise, the operation is unpredictable.

Example

Copy the samples from the `/BeTRAN` directory where OpenTP1 was installed to the OpenTP1 home directory (using the `cp` command):

```
% cp -R /BeTRAN/examples $DCDIR <CR>
```

An environment that allows the OpenTP1 samples to be used is now established.

8.2.2 Tasks specific to the Base sample (when using a stub)

This subsection explains how to use the Base sample. It assumes that the sample is used under the following conditions:

Shell to be used: C shell

OpenTP1 home directory: `/usr/betran`

(1) Creating application programs

Given below is the procedure for creating UAPs of the Base sample. To create UAPs of the Base sample, use the `make` command, a UNIX tool. A `makefile` dedicated to the sample is provided in the directory for the TP1/Server Base sample.

The `make` command should be executed on the assumption that the `c/` or `cobol/` directory in the `aplib/` directory is the current directory.

For example, to create a UAP in the C language, enter the commands as follows:

```
% chdir $DCDIR/examples/base/aplib/c <CR>
% make <CR>
```

When these commands are executed, executable UAP files (in C) with the names `basespp` and `basesup` are created in the `aplib/` directory.

(2) Modifying the system definition

The sample provides a system definition sample so that the user need not specify a system definition. For some definition files, however, the actual OpenTP1 home directory must be specified with the full pathname.

(a) Procedure for modifying the definition of the OpenTP1 home directory

The `chconf` command, a tool for modification, is used to change the OpenTP1 home directory from `$DCDIR` to the actual home directory. By executing this command, the specifications of the OpenTP1 home directory in the definition file can be changed from `$DCDIR` to the actual OpenTP1 home directory (`/usr/betran`, for example).

Before the `chconf` command can be executed, the user must go to the `$DCDIR/example/base/conf/` directory of the sample. A command input example is given below.

```
% chdir $DCDIR/examples/base/conf <CR>
% chconf <CR>
```

When the `chconf` command is executed, the contents of the following definition files are modified. The characters in bold represent the actual OpenTP1 home directory.

Table 8-1: Definition files and content to be modified (Base sample)

Definition file to be modified	Modification
`env`	`putenv DCCONFPATH $DCDIR/examples/base/conf`
`prc`	`putenv prcsvpath $DCDIR/examples/base/aplib`
`sts`	Physical file name: `$DCDIR/examples/base/betranfile/`*xxx*
`sysjnl`	Physical file name: `$DCDIR/examples/base/betranfile/`*xxx*

Definition file to be modified	Modification
cdtrn	Physical file name: $DCDIR/examples/base/betranfile/*xxx*

Before this tool (chconf command) can be executed, the OpenTP1 home directory must be defined in the environment variable DCDIR. Otherwise, the result is unpredictable.

(b) How to restore the modified OpenTP1 home directory

To restore the modified OpenTP1 home directory to its original state, execute the bkconf command provided by the sample. This command restores the portion in the definition file that was modified by the chconf command to its initial state.

If the chconf command fails to modify the system definition, execute the bkconf command at once.

A command input example is given below.

```
% chdir $DCDIR/examples/base/conf <CR>
% bkconf <CR>
```

(3) Setting environment variables and definition files

Given below is the procedure for starting the OpenTP1 system with the created sample UAPs and sample system definition.

(a) Set the environment variable DCCONFPATH

Set the directory containing the definition files in the environment variable DCCONFPATH. After this setting, OpenTP1 can recognize the contents of the definition files.

A command input example is given below:

```
% setenv DCCONFPATH $DCDIR/examples/base/conf <CR>
```

(b) Copy the definition file env

Of definition files, only the env file must be read from $DCDIR/conf into OpenTP1. Therefore, the env definition file created as a sample must be moved to $DCDIR/conf.

If an env definition file is already in $DCDIR/conf/ directory, it is overwritten.

Save it if necessary.

A command input example is given below:

```
% cp $DCDIR/examples/base/conf/env $DCDIR/conf <CR>
```

(c) Initialize the OpenTP1 file system

Initialize the OpenTP1 file system for Base sample using the shell file `base_mkfs`.

A command input example is given below:

```
% base_mkfs <CR>
```

After this shell file is executed, a file named `betranfile` is created under the `$DCDIR/examples/base/` directory and the OpenTP1 file system is established under that file.

8.2.3 Tasks for using OpenTP1 (when using a stub)

After modifying the system definition, start with the work for using OpenTP1.

(1) Setting up OpenTP1

Setup an OpenTP1 using the `dcsetup` command. The `dcsetup` command is placed in the `/BeTRAN/bin/` directory.

A command input example is given below:

```
% /BeTRAN/bin/dcsetup OpenTP1-home-directory-name <CR>
```

The OpenTP1 administrator is responsible for this work. Specify the full pathname with the `dcsetup` command only when using the sample for the first time. It is unnecessary to execute the `dcsetup` command specifying the full pathname to setup the sample again. For details on the `dcsetup` command, see the manual *OpenTP1 Operation*.

(2) Activating the OpenTP1 system and user server

The procedure for activating the OpenTP1 system and user server is given below.

(a) Start the OpenTP1 system

Start the OpenTP1 system using the `dcstart` command.

A command input example is given below:

```
% dcstart <CR>
```

(b) Start user servers (UAPs)

Start the created UAPs using the `dcsvstart` command. First start the server UAP (SPP), then start the client UAP (SUP).

A command input example is given below:

```
% dcsvstart -u basespp <CR>
```

A message log is output to indicate that basespp becomes online.

```
% dcsvstart -u basesup <CR>
```

A message log is output to indicate that basesup becomes online.

How the user server process proceeds is indicated by the message log.

The server UAP (SPP) can also be started automatically when the OpenTP1 system starts if so specified in the user service configuration definition.

(3) List of files in OpenTP1 file system

After the base_mkfs command, an OpenTP1 file system creation tool, is executed, an OpenTP1 file system is created under the $DCDIR/examples/base/betranfile file. The table below lists the files contained in the created OpenTP1 file system.

Table 8-2: List of files in OpenTP1 file system (Base sample)

File name	Purpose	Record length#	Number of records
jn101	System journal file	4096 bytes	50
jn102	System journal file	4096 bytes	50
jn103	System journal file	4096 bytes	50
stsfi101	Status file	4608 bytes	50
stsfi102	Status file	4608 bytes	50
stsfi103	Status file	4608 bytes	50
stsfi104	Status file	4608 bytes	50
cpdf01	Checkpoint dump file	4096 bytes	256
cpdf02	Checkpoint dump file	4096 bytes	256
cpdf03	Checkpoint dump file	4096 bytes	256

#

This record length is a default value.

(4) Exchanging a sample UAP

To exchange a sample UAP:

1. Terminate the OpenTP1 system.
2. Execute the dcsetup command with -d option to remove the OpenTP1 from the OS temporarily.

3. Specify the desired sample UAP by following the procedure in *8.2 How to use Base sample*.
4. Execute the UAP.

8.2.4 Tasks specific to the Base sample (when using dynamic loading of service functions)

This subsection describes the preparation procedure specific to the Base sample when using dynamic loading of service functions. The description assumes that the sample is used under the following conditions:

Shell to be used: C shell

OpenTP1 home directory: `/usr/betran`

(1) Creating application programs

The procedure for creating UAPs from the Base sample when using dynamic loading of service functions is given below. To create a sample program, use the UNIX `make` command. A `makefile` for the sample is provided in the directory for the Base sample.

The `make` command should be executed with the current directory set to the `c/` or `cobol/` directory in the `aplib/` directory.

For example, to create a UAP in C language, enter the commands as follows:

```
% chdir $DCDIR/examples/base/aplib/c <CR>
% make -f make_svdl <CR>
```

When these commands are executed, executable UAP files (in C) with the names `basespp2` and `basesup2` are created in the `aplib/` directory.

(2) Modifying the system definition

A sample definition file is provided with the sample, to save the user the trouble of modifying the system definition. However, for some definition files, the actual OpenTP1 home directory must be specified as an absolute path.

(a) Procedure for modifying the definition of the OpenTP1 home directory

The `chconf` command, a configuration tool, is used to change the OpenTP1 home directory from `$DCDIR` to the actual home directory. By executing this command, the references to the OpenTP1 home directory in the definition file can be changed from the placeholder `$DCDIR` to the actual OpenTP1 home directory (`/usr/betran`, for example).

Before executing the `chconf` command, navigate to the `$DCDIR/examples/base/conf/` directory. A command input example is given below:

```
% chdir $DCDIR/examples/base/conf <CR>
% chconf <CR>
```

When the chconf command is executed, the contents of the following definition files are modified. The portion in bold is changed to the actual OpenTP1 home directory.

Table 8-3: Definition files and content to be modified (Base sample)

Definition file to be modified	Modification
env	putenv DCCONFPATH $DCDIR/examples/base/conf
prc	putenv prcsvpath $DCDIR/examples/base/aplib
sts	Physical file name: $DCDIR/examples/base/betranfile/*xxx*
sysjnl	Physical file name: $DCDIR/examples/base/betranfile/*xxx*
cdtrn	Physical file name: $DCDIR/examples/base/betranfile/*xxx*

Before this tool (chconf command) can be executed, the OpenTP1 home directory must be defined in the DCDIR environment variable. Otherwise, the files are not modified correctly.

Because the UAP shared library name (containing $DCDIR) specified in the service operand of basespp2 and BASESPP2 is specified using an environment variable, executing the chconf command will not change references to this file in the definition files.

(b) Restoring the modified OpenTP1 home directory

To restore the modified OpenTP1 home directory to its original state, execute the bkconf command provided to undo the change. This command restores the parts of the definition files modified by the chconf command to their initial state.

If the chconf command fails to modify the system definition in the manner expected, immediately execute the bkconf command.

A command input example is given below:

```
% chdir $DCDIR/examples/base/conf <CR>
% bkconf <CR>
```

(3) Setting environment variables and definition files

The procedure for starting the OpenTP1 system with the created sample UAPs and sample system definition is given below.

(a) Set the environment variable DCCONFPATH

Set the directory containing the definition files as the value of the environment variable DCCONFPATH. This allows OpenTP1 to recognize the contents of the definition files.

A command input example is given below:

```
% setenv DCCONFPATH $DCDIR/examples/base/conf <CR>
```

(b) Copy the definition file env

Of the definition files, only the env file must be read from $DCDIR/conf into OpenTP1. Therefore, the env definition file created as a sample must be moved to $DCDIR/conf.

If an env definition file has already been created in the $DCDIR/conf/ directory, it will be overwritten. Back up the existing file if necessary.

A command input example is given below:

```
% cp $DCDIR/examples/base/conf/env $DCDIR/conf <CR>
```

(c) Initialize the OpenTP1 file system

Initialize the OpenTP1 file system for the Base sample by executing the shell file base_mkfs.

A command input example is given below:

```
% base_mkfs <CR>
```

When this shell file is executed, a file named betranfile is created under the $DCDIR/examples/base/ directory and the OpenTP1 file system is established under that file.

8.2.5 Tasks for using OpenTP1 (when using dynamic loading of service functions)

After modifying the system definition, begin the tasks required for using OpenTP1.

(1) Setting up OpenTP1

Set up OpenTP1 by executing the dcsetup command. The dcsetup command is located in the /BeTRAN/bin/ directory.

A command input example is given below:

```
% /BeTRAN/bin/dcsetup OpenTP1-home-directory-name <CR>
```

The setup task must be performed by an OpenTP1 system administrator. Specify the dcsetup command as an absolute path only when using the sample for the first time. You do not need to specify the absolute path for the dcsetup command the next time you set up the sample. For details on the dcsetup command, see the manual *OpenTP1 Operation*.

(2) Activating the OpenTP1 system and user servers

The procedure for activating the OpenTP1 system and user servers is given below.

(a) Start the OpenTP1 system

Start the OpenTP1 system by using the dcstart command.

A command input example is given below:

```
% dcstart <CR>
```

(b) Start user servers (UAPs)

Start the created UAPs using the dcsvstart command. Start the server UAP (SPP) first, and then the client UAP (SUP).

A command input example is given below:

```
% dcsvstart -u basespp2 <CR>
```

A message log is output to indicate that basespp2 is online.

```
% dcsvstart -u basesup2 <CR>
```

A message log is output to indicate that basesup2 is online.

The processing activity of the user servers (UAPs) is output to the message log.

The server UAP (SPP) can also be set to start automatically when the OpenTP1 system starts by making the appropriate setting in the user service configuration definition.

(3) List of files in the OpenTP1 file system

When you execute the base_mkfs command to create the OpenTP1 file system, the file system is created under the $DCDIR/examples/base/betranfile file. The table below lists the files contained in the created OpenTP1 file system.

Table 8-4: List of files in the OpenTP1 file system (Base sample)

File name	Purpose	Record length#	Number of records
jnl01	System journal file	4096 bytes	50

File name	Purpose	Record length#	Number of records
jnl02	System journal file	4096 bytes	50
jnl03	System journal file	4096 bytes	50
stsfil01	Status file	4608 bytes	50
stsfil02	Status file	4608 bytes	50
stsfil03	Status file	4608 bytes	50
stsfil04	Status file	4608 bytes	50
cpdf01	Checkpoint dump file	4096 bytes	256
cpdf02	Checkpoint dump file	4096 bytes	256
cpdf03	Checkpoint dump file	4096 bytes	256

#

These record lengths are default values.

(4) Replacing a sample UAP

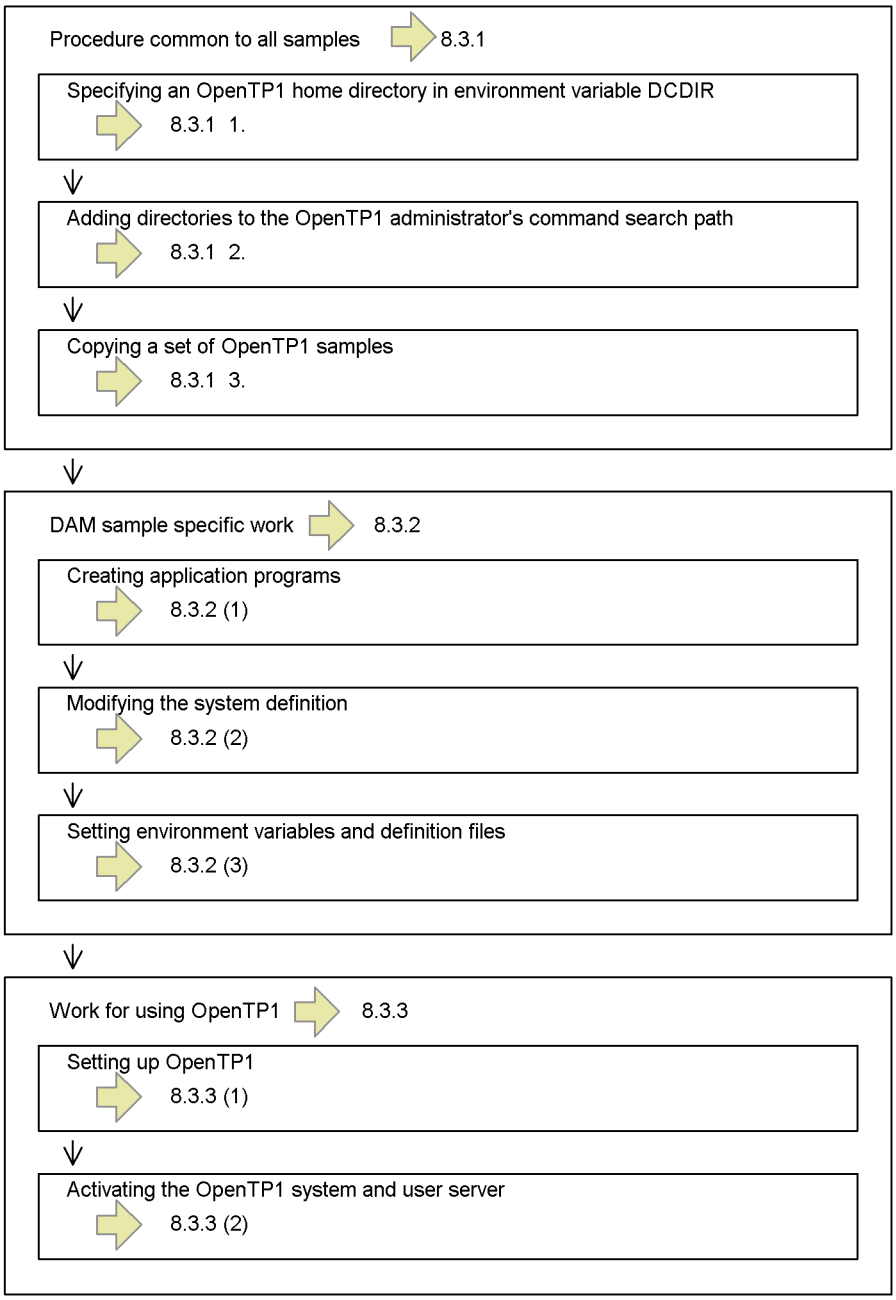
To replace a sample UAP:

1. Shut down the OpenTP1 system.
2. Execute the `dcsetup` command with the `-d` option specified to temporarily remove OpenTP1 from the OS.
3. Specify the new sample UAP that you want to use by following the procedure in *8.2 How to use Base sample*.
4. Execute the UAP.

8.3 How to use DAM sample

This section explains how to use the DAM sample. The figure below shows the outline of the procedure for using samples.

Figure 8-4: Outline of procedure for using samples (DAM sample)



8.3.1 Procedure common to all samples (DAM sample)

The following preparation procedure common to three OpenTP1 samples is required:

1. Specifying an OpenTP1 home directory in environment variable `DCDIR`
2. Adding directories to the OpenTP1 administrator's command search path
3. Copying a set of OpenTP1 samples

For details on the above procedure, see *8.2.1 Procedure common to all samples (Base sample)*.

8.3.2 DAM sample specific work

This subsection explains how to use the DAM sample. It assumes that the sample is used under the following conditions:

Shell to be used: C shell

OpenTP1 home directory: `/usr/betran`

(1) Creating application programs

Given below is the procedure for creating UAPs of the DAM sample. To create UAPs of the DAM sample, use the `make` command, a UNIX tool. A `makefile` dedicated to the sample is provided in the directory for the DAM sample.

(a) Create an object file for transaction control

The OpenTP1 command `trnmkobj` is used to create an object file for transaction control.

The `makefile` used with the sample specifies that the object file for transaction control be created under the name `dam_sw.o`. Therefore, execute the `trnmkobj` command to create an object file so that the created object file is named `dam_sw.o`.

The object file is created under the `$DCDIR/spool/trnrmcmd/userobj/` directory. If an object file of the same name already exists in this directory, save it in advance.

A command input example is given below:

```
% trnmkobj -o dam_sw -R OpenTP1_DAM <CR>
```

(b) Create a UAP executable file

Execute the `make` command on the assumption that the `c/` or `cobol/` directory in the `aplib/` directory is the current directory.

For example, to create a UAP in C, enter the commands as follows:

```
% chdir $DCDIR/examples/dam/aplib/c <CR>
% make <CR>
```

When these commands are executed, a UAP executable file (in C) is created in the `aplib/` directory.

(2) Modifying the system definition

The sample provides a system definition sample so that the user need not specify a system definition. For some definition files, however, the actual OpenTP1 home directory must be specified with the full pathname.

(a) Procedure for modifying the definition of the OpenTP1 home directory

The `chconf` command, a tool for modification, is used to change the OpenTP1 home directory from `$DCDIR` to the actual home directory. By executing this command, the specifications of the OpenTP1 home directory in the definition file can be changed from `$DCDIR` to the actual OpenTP1 home directory (`/usr/betran`, for example).

Before the `chconf` command can be executed, the user must go to the `conf/` directory of the sample.

A command input example is given below:

```
% chdir $DCDIR/examples/dam/conf <CR>
% chconf <CR>
```

When the `chconf` command is executed, the contents of the following definition files are modified. The characters in bold represent the portion to be modified.

Table 8-5: Definition files and content to be modified (DAM sample)

Definition file to be modified	Modification
`env`	`putenv DCCONFPATH $DCDIR/examples/dam/conf`
`prc`	`putenv prcsvpath $DCDIR/examples/dam/aplib`
`sts`	Physical file name: `$DCDIR/examples/dam/betranfile/`*xxx*
`sysjnl`	Physical file name: `$DCDIR/examples/dam/betranfile/`*xxx*
`cdtrn`	Physical file name: `$DCDIR/examples/dam/betranfile/`*xxx*

Before this tool (`chconf` command) can be executed, the OpenTP1 home directory must be defined in the environment variable `DCDIR`. Otherwise, the result is unpredictable.

(b) How to restore the modified OpenTP1 home directory to its original state

To restore the modified OpenTP1 home directory to its original state, execute the `bkconf` command provided by the sample. This command restores the portion in the definition file that was modified by the `chconf` command to its initial state.

If the chconf command fails to modify the system definition, execute the bkconf command at once.

A command input example is given below:

```
% chdir $DCDIR/examples/dam/conf <CR>
% bkconf <CR>
```

(3) Setting environment variables and definition files

Given below is the procedure for starting the OpenTP1 system with the created sample UAPs and sample system definition.

(a) Set the environment variable DCCONFPATH

Set the directory containing the definition files in the environment variable DCCONFPATH. After this setting, OpenTP1 can recognize the contents of the definition files.

A command input example is given below:

```
% setenv DCCONFPATH $DCDIR/examples/dam/conf <CR>
```

(b) Copy the definition file env

Of definition files, only the env file must be read from $DCDIR/conf into OpenTP1. Therefore, the env definition file created as a sample must be moved to $DCDIR/conf.

If an env definition file is already in $DCDIR/conf/ directory, it is overwritten. Save it if necessary.

A command input example is given below:

```
% cp $DCDIR/examples/dam/conf/env $DCDIR/conf <CR>
```

(c) Initialize the OpenTP1 file system

Initialize the OpenTP1 file system for DAM sample using the shell file dam_mkfs.

A command input example is given below:

```
% dam_mkfs <CR>
```

After this shell file is executed, a file named betranfile is created under the $DCDIR/examples/dam/ directory and the OpenTP1 file system is established under that file.

8.3.3 Work for using OpenTP1

After modifying the system definition, start with the work for using OpenTP1.

(1) Setting up OpenTP1

Setup an OpenTP1 using the `dcsetup` command. The `dcsetup` command is placed in the `/BeTRAN/bin/` directory.

A command input example is given below:

```
% /BeTRAN/bin/dcsetup OpenTP1-home-directory-name <CR>
```

The OpenTP1 administrator is responsible for this work. Specify the full path name with the `dcsetup` command only when using the sample for the first time. It is unnecessary to execute the `dcsetup` command specifying the full pathname to setup the sample again. For details on the `dcsetup` command, see the manual *OpenTP1 Operation*.

(2) Activating the OpenTP1 system and user server

The procedure for activating the OpenTP1 system by using the created sample UAP and system definition for the sample is given below.

(a) Start the OpenTP1 system

Start the OpenTP1 system using the `dcstart` command.

A command input example is given below:

```
% dcstart <CR>
```

(b) Start user servers (UAPs)

Start the created UAPs using the `dcsvstart` command. First start the server UAP (SPP), then start the client UAP (SUP).

A command input example is given below:

```
% dcsvstart -u damspp <CR>
```

A message log is output to indicate that `damspp` becomes online.

```
% dcsvstart -u damsup <CR>
```

A message log is output to indicate that `damsup` becomes online.

How the user server process proceeds is indicated by the message log.

The server UAP (SPP) can also be activated automatically when the OpenTP1 system starts if so specified in the user service configuration definition.

(3) List of files in OpenTP1 file system

After the `dam_mkfs` command, an OpenTP1 file system creation tool, is executed, an OpenTP1 file system is created under the `$DCDIR/examples/dam/betranfile/`

directory. The table below lists the files contained in the created OpenTP1 file system.

Table 8-6: List of files in OpenTP1 file system (DAM sample)

File name	Purpose	Record length#	Number of records
jn1f01	System journal file	4096 bytes	100
jn1f02	System journal file	4096 bytes	100
jn1f03	System journal file	4096 bytes	100
stsfi101	Status file	4608 bytes	64
stsfi102	Status file	4608 bytes	64
stsfi103	Status file	4608 bytes	64
stsfi104	Status file	4608 bytes	64
cpdf01	Checkpoint dump file	4096 bytes	100
cpdf02	Checkpoint dump file	4096 bytes	100
cpdf03	Checkpoint dump file	4096 bytes	100
smplfile	DAM file	512 bytes#	11 blocks

#

This value indicates the DAM file block length.

(4) Exchanging a sample UAP

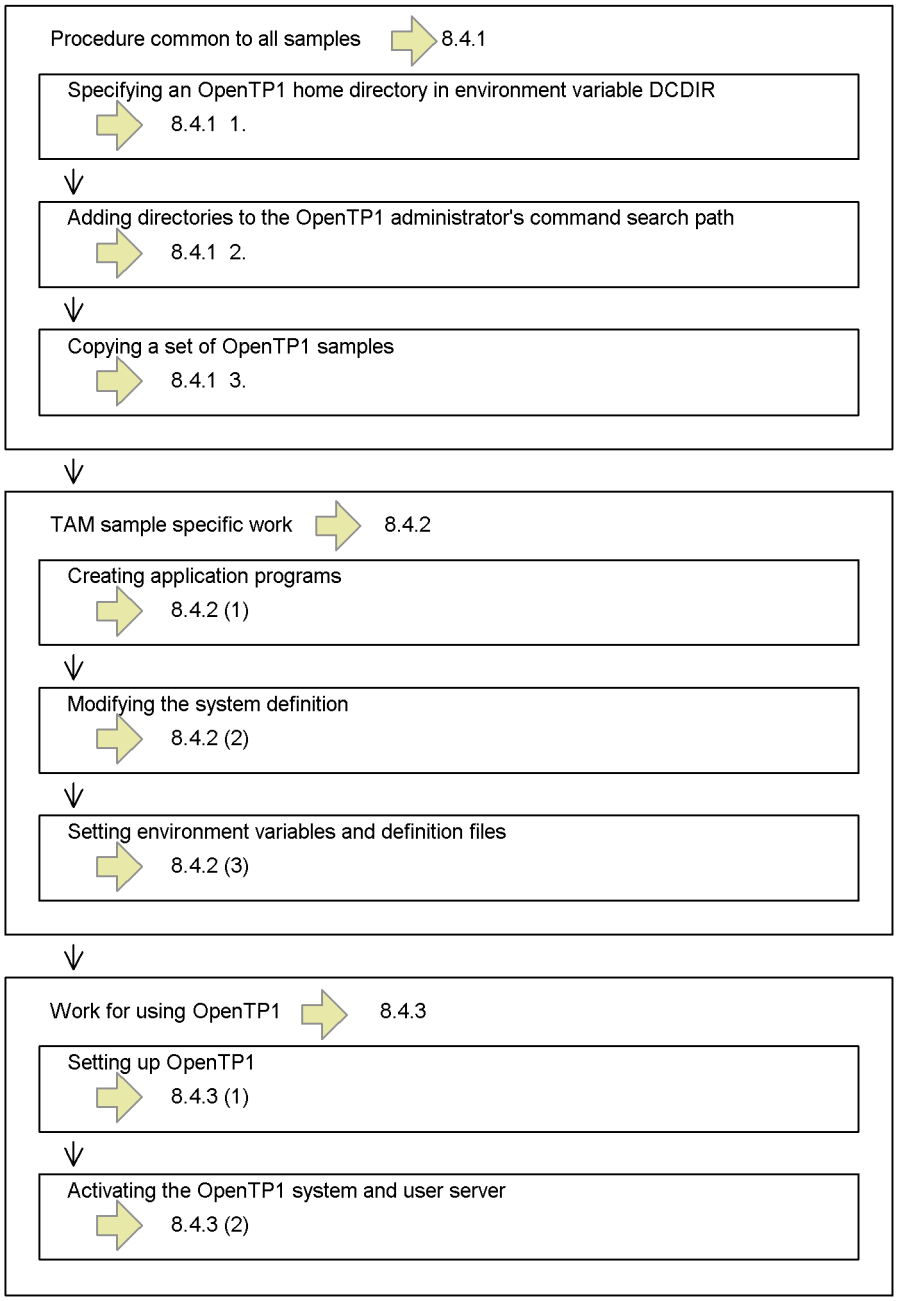
To exchange a sample UAP:

1. Terminate the OpenTP1 system.
2. Execute the `dcsetup` command with `-d` option to remove the OpenTP1 from the OS temporarily.
3. Specify the desired sample UAP following the procedure in *8.3 How to use DAM sample*.
4. Execute the UAP.

8.4 How to use TAM sample

This section explains how to use the TAM sample. The figure below shows the outline of the procedure for using samples.

Figure 8-5: Outline of procedure for using samples (TAM sample)



8.4.1 Procedure common to all samples (TAM sample)

The following preparation procedure common to three OpenTP1 samples is required:

1. Specifying an OpenTP1 home directory in environment variable `DCDIR`
2. Adding directories to the OpenTP1 administrator's command search path
3. Copying a set of OpenTP1 samples

For details on the above procedure, see *8.2.1 Procedure common to all samples (Base sample)*.

8.4.2 TAM sample specific work

This subsection explains how to use the TAM sample. It assumes that the sample is used under the following conditions:

Shell to be used: C shell

OpenTP1 home directory: `/usr/betran`

(1) Creating application programs

Given below is the procedure for creating UAPs of the TAM sample. To create UAPs of the TAM sample, use the `make` command, a UNIX tool. A `makefile` dedicated to the sample is provided in the directory for the TAM sample.

(a) Create an object file for transaction control

The OpenTP1 command `trnmkobj` is used to create an object file for transaction control.

The `makefile` used with the sample specifies that the object file for transaction control be created under the name `tam_sw.o`. Therefore, execute the `trnmkobj` command to create an object file so that the created object file is named `tam_sw.o`.

The object file is created under the `$DCDIR/spool/trnrmcmd/userobj/` directory. If an object file of the same name already exists in this directory, save it in advance.

A command input example is given below:

```
% trnmkobj -o tam_sw -R OpenTP1_TAM <CR>
```

(b) Create a UAP executable file

Execute the `make` command on the assumption that the `c/` or `cobol/` directory in the `aplib/` directory is the current directory. For example, to create a UAP in C, enter the commands as follows:

```
% chdir $DCDIR/examples/tam/aplib/c <CR>
% make <CR>
```

When these commands are executed, a UAP executable file (in C) is created in the aplib/ directory.

(2) Modifying the system definition

The sample provides a system definition initial values to help the user specify a system definition. For some definition files, however, the actual OpenTP1 home directory must be specified with the full pathname.

(a) Procedure for modifying the definition of the OpenTP1 home directory

The chconf command, a tool for modification, is used to change the OpenTP1 home directory from $DCDIR to the actual home directory. By executing this command, the specifications of the OpenTP1 home directory in the definition file can be changed from $DCDIR to the actual OpenTP1 home directory (/usr/betran, for example).

Before the chconf command can be executed, the user must go to the conf/ directory of the sample. A command input example is given below:

```
% chdir $DCDIR/examples/tam/conf <CR>
% chconf <CR>
```

When the chconf command is executed, the contents of the following definition files are modified. The characters in bold represent the portion to be modified.

Table 8-7: Definition files and content to be modified (TAM sample)

Definition file to be modified	Modification
env	putenv DCCONFPATH $DCDIR/examples/tam/conf
prc	putenv prcsvpath $DCDIR/examples/tam/aplic
sts	Physical file name: $DCDIR/examples/tam/betranfile/*xxx*
sysjnl	Physical file name: $DCDIR/examples/tam/betranfile/*xxx*
cdtrn	Physical file name: $DCDIR/examples/tam/betranfile/*xxx*

Before this tool (chconf command) can be executed, the OpenTP1 home directory must be defined in the environment variable DCDIR. Otherwise, the result is unpredictable.

(b) How to restore the modified OpenTP1 home directory to its original state

To restore the modified OpenTP1 home directory to its original state, execute the bkconf command provided by the sample. This command restores the portion in the definition file that was modified by the chconf command to its initial state.

If the chconf command fails to modify the system definition, execute the bkconf command at once.

A command input example is given below:

```
% chdir $DCDIR/examples/tam/conf <CR>
% bkconf <CR>
```

(3) Setting environment variables and definition files

Given below is the procedure for starting the OpenTP1 system with the created sample UAPs and sample system definition.

(a) Set the environment variable DCCONFPATH

Set the directory containing the definition files in the environment variable `DCCONFPATH`. After this setting, OpenTP1 can recognize the contents of the definition files.

A command input example is given below:

```
% setenv DCCONFPATH $DCDIR/examples/tam/conf <CR>
```

(b) Copy the definition file env

Of definition files, only the `env` file must be read from `$DCDIR/conf` into OpenTP1. Therefore, the `env` definition file created as a sample must be moved to `$DCDIR/conf`.

If an `env` definition file is already in the `$DCDIR/conf/` directory, it is overwritten. Save it if necessary.

A command input example is given below:

```
% cp $DCDIR/examples/tam/conf/env $DCDIR/conf <CR>
```

(c) Initialize the OpenTP1 file system

Initialize the OpenTP1 file system for TAM sample using the shell file `tam_mkfs`.

A command input example is given below:

```
% tam_mkfs <CR>
```

After this shell file is executed, a file named `betranfile` is created under the `$DCDIR/examples/tam/` directory and the OpenTP1 file system is established under that file.

8.4.3 Work for using OpenTP1

After modifying the system definition, start with the work for using OpenTP1.

(1) Setting up OpenTP1

Setup an OpenTP1 using the `dcsetup` command. The `dcsetup` command is placed

in the /BeTRAN/bin/ directory.

A command input example is given below:

```
% /BeTRAN/bin/dcsetup OpenTP1-home-directory-name <CR>
```

The OpenTP1 administrator is responsible for this work. Specify the full pathname with the dcsetup command only when using the sample for the first time. It is unnecessary to execute the dcsetup command specifying the full pathname to setup the sample again. For details on the dcsetup command, see the manual *OpenTP1 Operation*.

(2) Activating the OpenTP1 system and user server

The procedure for activating the OpenTP1 system by using the created sample UAP and system definition for the sample is given below.

(a) Start the OpenTP1 system

Start the OpenTP1 system using the dcstart command. A command input example is given below:

```
% dcstart <CR>
```

(b) Start user servers (UAPs)

Start the created UAPs using the dcsvstart command. First start the server UAP (SPP), then start the client UAP (SUP). A command input example is given below:

```
% dcsvstart -u tamspp <CR>
```

A message log is output to indicate that tamspp becomes online.

```
% dcsvstart -u tamsup <CR>
```

A message log is output to indicate that tamsup becomes online.

How the user server process proceeds is indicated by the message log.

The server UAP (SPP) can also be activated automatically when the OpenTP1 system starts if so specified in the user service configuration definition.

(3) List of files in OpenTP1 file system

After the tam_mkfs command, an OpenTP1 file system creation tool, is executed, an OpenTP1 file system is created under the $DCDIR/examples/tam/betranfile directory. The table below lists the files contained in the created OpenTP1 file system.

Table 8-8: List of files in OpenTP1 file system (TAM sample)

File name	Purpose	Record length	Number of records
jnf101	System journal file	4096 bytes	50
jnf102	System journal file	4096 bytes	50
jnf103	System journal file	4096 bytes	50
stsfi101	Status file	4608 bytes	256
stsfi102	Status file	4608 bytes	256
stsfi103	Status file	4608 bytes	256
stsfi104	Status file	4608 bytes	256
cpdf01	Checkpoint dump file	4096 bytes	100
cpdf02	Checkpoint dump file	4096 bytes	100
cpdf03	Checkpoint dump file	4096 bytes	100

The table below lists the specifications of the created TAM file.

Table 8-9: Specifications of the TAM sample file

File name	tamexam1
Purpose	TAM file
Record length	40 bytes (including the key length)
Key area length	20 bytes
Key start position	0th byte (beginning of the record)
Maximum number of records	10
Table format	Tree
TAM data file name	$DCDIR/examples/tools/tamdata

(4) Exchanging a sample UAP

To exchange a sample UAP:

1. Terminate the OpenTP1 system.
2. Execute the dcsetup command with -d option to remove the OpenTP1 from the OS temporarily.
3. Specify the desired sample UAP following the procedure in *8.4 How to use TAM*

sample.

4. Execute the UAP.

8.5 Specifications of sample programs

This section explains the specifications of the following three OpenTP1 samples:

- Base sample
- DAM sample
- TAM sample

The specifications are common to the above samples.

8.5.1 Contents of database used by samples

The UAP references private information in the established customer information database using names as the key or updates sales amounts. The table below indicates the format of the customer information database.

Table 8-10: Format of customer information database

Name	Sex	Age	Sales amount
Tanaka	Male	25	200,000
Saitoh	Female	22	1,200,000
Nakamura	Male	30	500,000
Miyamoto	Male	19	800,000
Suzuki	Female	20	950,000

8.5.2 Outline of sample program processing

An outline of sample program processing is given below. For details, see the source file of the sample program.

The client UAP retrieves private information about one person by sending a reference request to the server. Then, it sends an update request to the server to update sales amounts. Finally, it sends a reference request to confirm that sales amounts have been updated.

When the client UAP requests the server UAP for service, operation can be checked through message log output. The message log is output after a reference if the request is a reference request or before a update if the request is an update request.

When a reference or update process is completed, a message log is also output to the server UAP to indicate whether the process was successful.

Messages output from the client UAP are given a character string `client` and

messages output from the server UAP are given a character string `server`. This helps identify which UAP issued the message. The figures below show the relationship between client and server UAP calls, for C and COBOL, respectively.

Figure 8-6: Relationship between client and server UAP calls (C language)

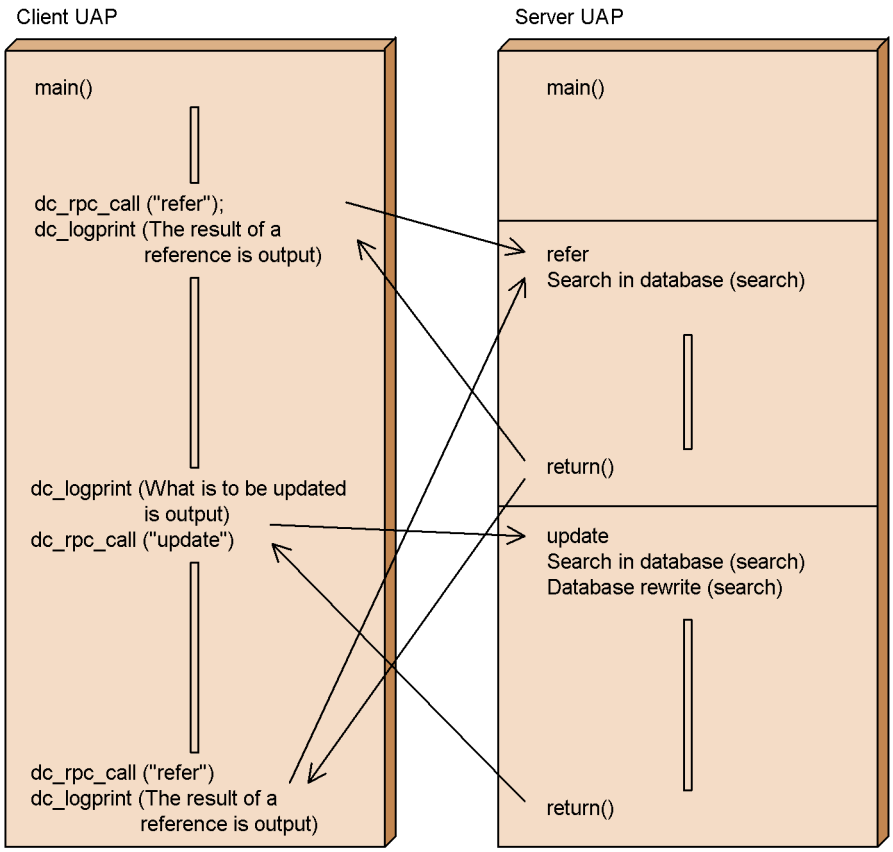
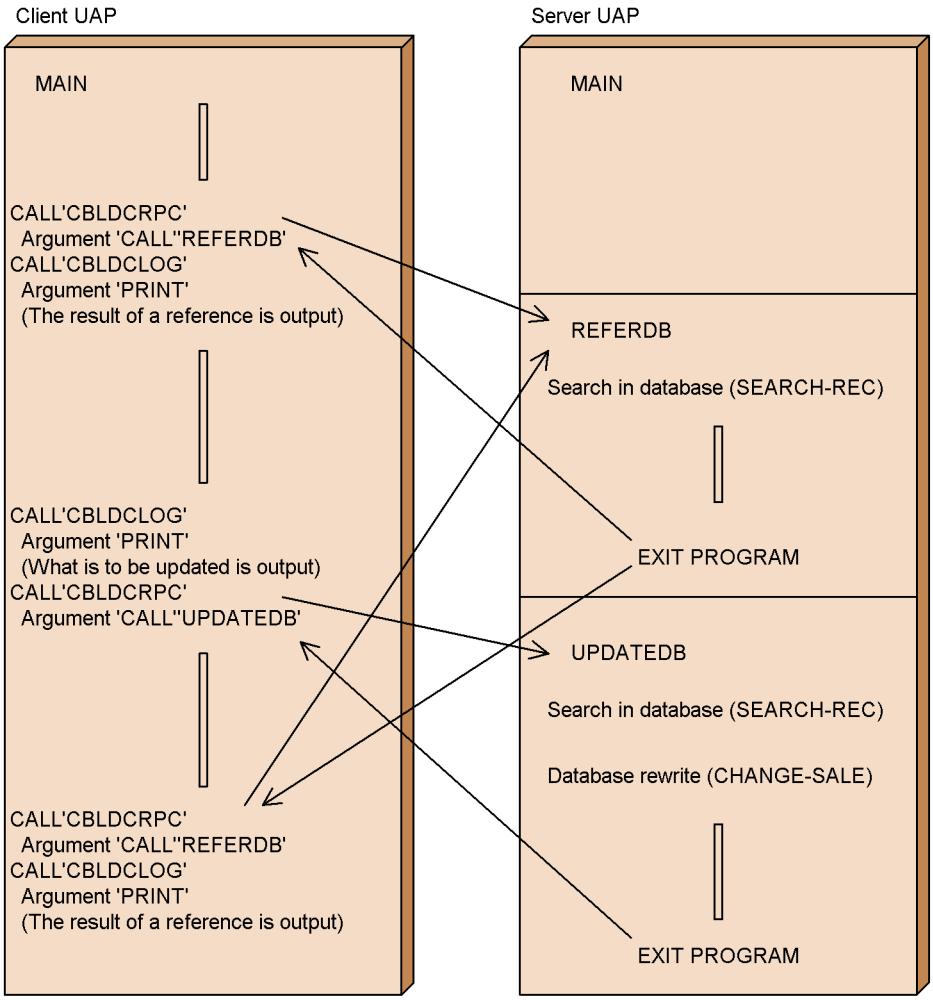


Figure 8-7: Relationship between client and server UAP calls (COBOL language)



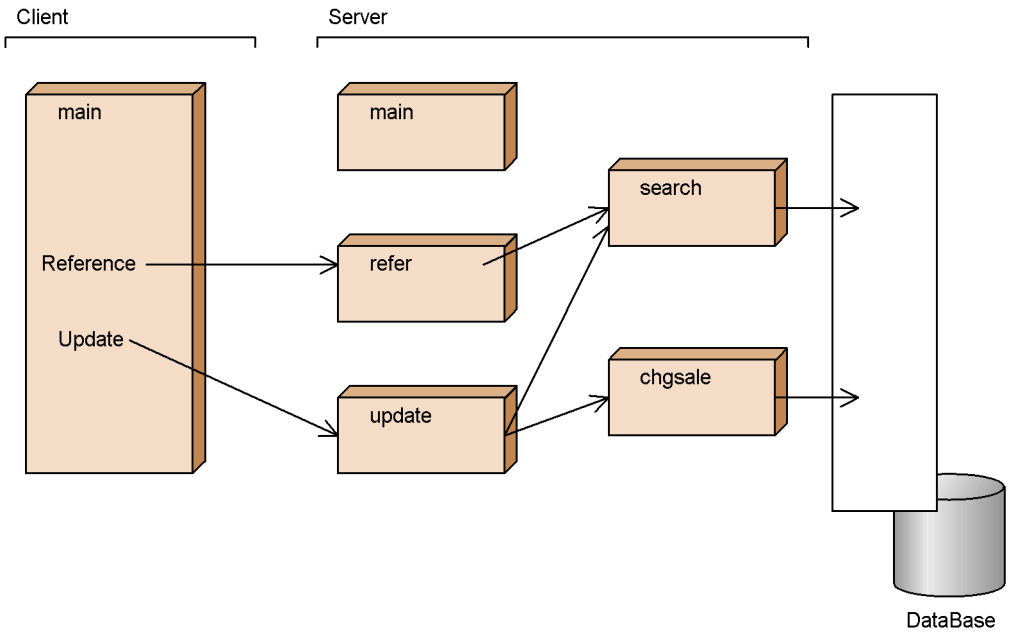
8.5.3 Structure of sample programs

The client UAP consists of a single program, whereas the server UAP consists of multiple programs. Some program names are different depending on whether the sample UAP is written in C or COBOL.

(1) Structure of programs written in C

The figure below shows the program structure of the client and server UAPs written in C.

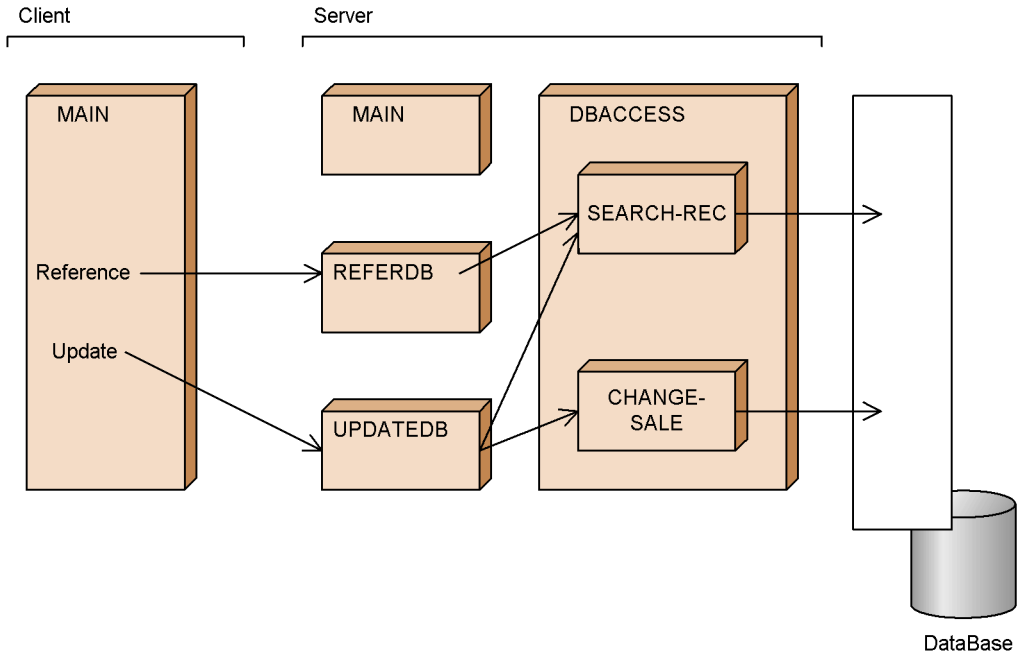
Figure 8-8: Program structure of client and server UAPs (C language)



(2) Structure of programs written in COBOL

The program structure of the server UAP written in COBOL contains an extra program. The figure below shows the program structure of the Base sample UAPs written in COBOL.

Figure 8-9: Program structure of client and server UAPs (COBOL language)



8.5.4 Details of programs specific to each sample

This subsection explains the specifications specific to each sample.

(1) Programs of Base sample

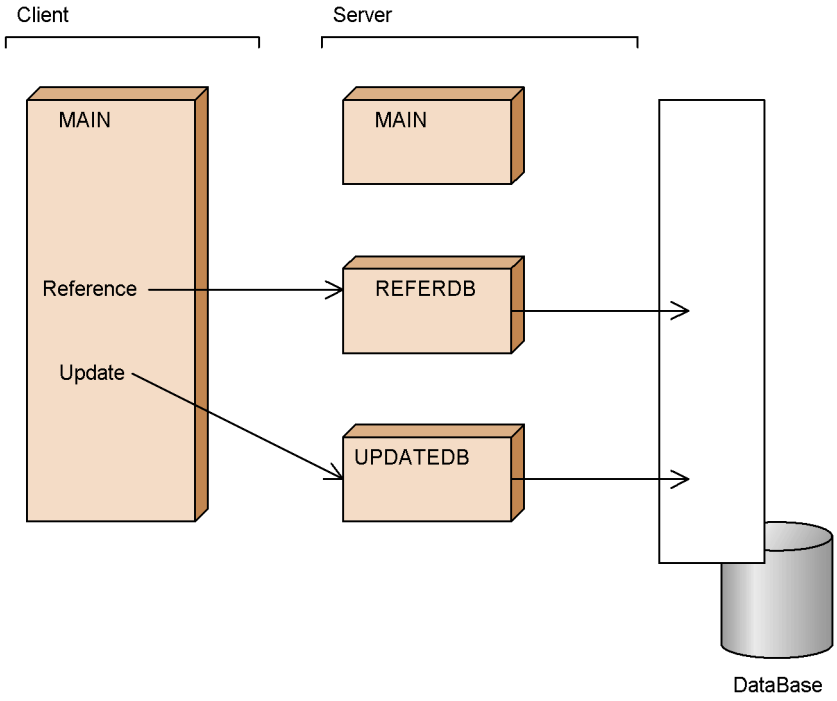
With the Base sample, the database (DataBase) is created inside the user server process and retained while the process is resident. Transaction start processing and synchronization point processing are performed on the server UAP during the update process.

(2) Programs of DAM sample

The programs of DAM sample are the same as the programs of Base sample except the following three points:

1. The program structure of the server UAP written in COBOL is different from the Base sample. The figure below shows the program structure of the DAM sample UAPs written in COBOL.

Figure 8-10: Program structure of client and server UAPs (DAM sample written in COBOL language)



2. Since the DAM sample creates a database (DataBase) using an offline program (`dam_mkfs`), the database is retained even after the user server process terminates.
3. Transaction start processing and synchronization point processing are performed on the server UAP during the process (reference or update).

(3) Programs of TAM sample

The programs of TAM sample are the same as the programs of Base sample except the following three points:

1. The program structure of the server UAP written in COBOL is different from the Base sample. For the program structure of the TAM sample UAPs written in COBOL, see Figure 8-10.
2. Since the TAM sample creates a database(DataBase) using an offline program (`tam_mkfs`), the database is retained even after the user server process terminates.
3. Transaction start processing and synchronization point processing are performed

on the server UAP during the process (reference or update).

(4) Notes on using sample programs

- When OpenTP1 is activated using sample programs, the `KFCA00901-W` message may appear. Ignore the message if it is about resource managers that are not used.
- The sample program `makefile` explicitly specifies `/bin/cc` as the C compiler. If `/bin/cc` does not exist or if a C compiler other than `/bin/cc` is used, specify the absolute path of the C compiler to be used on `makefile` before using the program.

8.6 How to use MCF sample

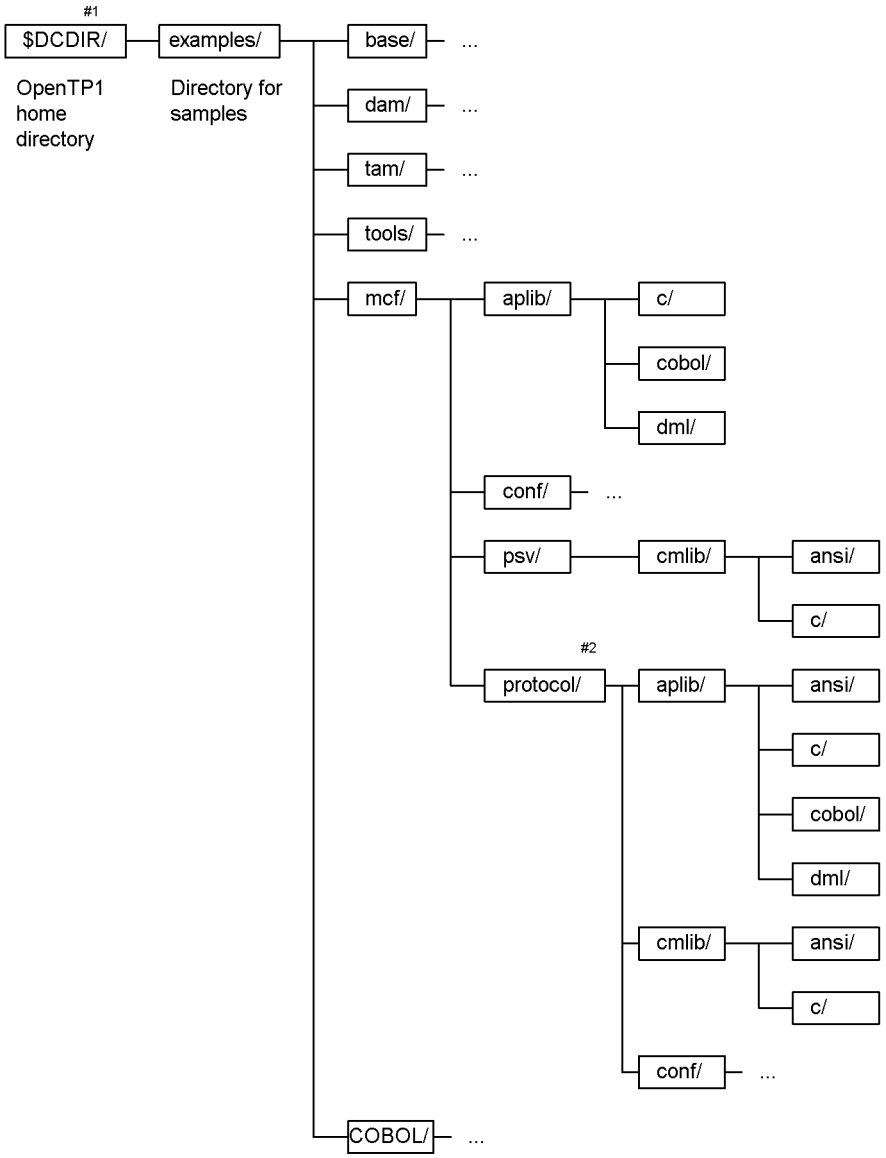
This section explains the sample of the message control facility (MCF) called MCF sample. The MCF sample enables the following examples on the manual to be used as UNIX text files:

- Example of system definition
- Coding example of MHP program (in C, COBOL, and COBOL with DML)
- Example of MCF main function (in ANSI C, C++, and K&R C style)

8.6.1 MCF sample directory configuration

The MCF sample is divided into an MCF basic section and a communication protocol section. The figure below shows the directories containing the MCF sample.

Figure 8-11: Configuration of directories for MCF sample



#1 $DCDIR is an environment variable indicating the OpenTP1 home directory. When a set of OpenTP1 samples is copied into the OpenTP1 home directory excluding /BeTRAN, this variable indicates the directory name. If not, the examples/ directory is placed under /BeTRAN.

#2 The name of the protocol/ directory is specific to the communication protocol supporting products.

Directories subordinate to the `$DCDIR/examples/mcf/` directory are listed below

with a brief explanation.

`aplib/`

Directory that contains MHP sample

`conf/`

Directory that contains system definition sample

`psv/cmlib/`

Directory that contains MCF main function sample

protocol/

Directory that contains sample specific to the communication protocol supporting product

The following directories are placed under the *protocol/* directory:

- Directory that contains MHP and SPP samples
- Directory that contains the system definition
- Directory that contains the MCF main function

(1) Contents of mcf/aplib/ directory

Files in the `mcf/aplib` directory are listed below with a brief explanation.

`c/`

Directory that contains MHP source files (in the C language). MHP main function (`mhp.c`) and MHP service function (`apl.c`) are stored here.

`cobol/`

Directory that contains MHP source files (in the COBOL language). MHP main function (`mhp.cbl`) and MHP service function (`ap.cbl`) are stored here.

`dml/`

Directory that contains MHP source files (in the COBOL language with DML). MHP service function (`ap.cbl`) is stored here.

For the contents of the above programs, see the coding example shown in the applicable *OpenTP1 Programming Reference* manual.

(2) Contents of conf/ directory

Files in the `mcf/conf/` directory are listed below with a brief explanation.

`abc_mngr`

Sample MCF manager definition

`abc_ua_c`

Sample common definition of MCF communication configuration definition. This definition is for OSAS/UA protocol.

`abc_ua_d`

Sample protocol-specific definition of MCF communication configuration definition. This definition is for OSAS/UA protocol.

`psvr_psvr_cmn`

Sample common definition of MCF communication configuration definition. This definition is for application starting definition.

`psvr_psvr_dta`

Sample application starting definition of MCF communication configuration definition.

`abc_apli`

Sample MCF application definition

`mcfu01`

Sample MCF system service information definition. This definition is the contents of the MCF communication process (for OSAS/UA).

`mcfu02`

Sample MCF system service information definition. This definition is the contents of the application communication process.

For the contents of the above definitions, see the definition sample shown in the manual *OpenTP1 System Definition*.

(3) Contents of psv/cmlib/ directory

Directories subordinate to the `mcf/psv/cmlib/` directory are listed below with a brief explanation.

`ansi/`

Sample MCF main function for application starting process (in ANSI C and C++ style)

`c/`

Sample MCF main function for application starting process (in K&R C style)

For the contents of the above MCF main functions, see the definition sample shown in the manual *OpenTP1 Operation*. For the MCF main function for MCF communication service, see the files in the `mcf/`*protocol*`/cmlib/` directory.

(4) Contents of protocol/ directory

Samples for each communication protocol supporting product are stored in the *protocol/* directory. The name of the *protocol/* directory depends on the communication protocol supporting product. The correspondence between the *protocol/* directory names and the product names is as follows:

`HDLC`: TP1/NET/HDLC

`HNANIF`: TP1/NET/HNA-NIF

`OSITP`: TP1/NET/OSI-TP

`TCPIP`: TP1/NET/TCP/IP

`XMAP3`: TP1/NET/XMAP3

`X25`: TP1/NET/X25

For the contents of each directory, see the applicable *OpenTP1 Protocol* manual.

8.6.2 Notes on using MCF sample

Notes on using the MCF sample are as follows:

1. When the definition relating to MCF (network communication definition) is used, the definition should be consistent in the MCF sample. Since the contents of system definition are exactly as in the manual, some modification may be required. The definition of TP1/Server Base (system service definition) should be modified according to the network communication definition of the MCF sample. This is the same when using the Base sample.
2. When the sample UAP (MHP or SPP) in the *protocol/* directory is used, a main function (main program) is required at compilation and link-edit. For MHPs, modify and use the main function in the `mcf/aplib/` directory. For SPPs, create a main function.

   The MCF application definition (`abc_apli`) in the `mcf/conf/` directory should be modified according to an MHP to be created. The system service definition (such as user service definition) should be created.
3. Use the following samples as an MHP main function:
   - For MCF main function for MCF communication service:

     File in the `/mcf/`*protocol*`/cmlib/` directory
   - For MCF main function for application starting service:

     File in the `/mcf/psv/cmlib/` directory

8.7 Samples to be used to dispatch multi OpenTP1 command

When a command is executed at a multi OpenTP1 node from another node using `rsh` or something similar, it cannot be determined which OpenTP1 system the command is executed in. In this case, a command must be executed by specifying the node name. A sample includes a command that executes a certain command specifying a node name (shell file). This command named `delvcmd` is stored in the `tools/` directory of Base sample.

(1) How to use delvcmd command

The `delvcmd` command is executed with the following syntax:

```
delvcmd -w node-name [, node-name] ... command-name
```

The *node-name* fields specify the identifiers of nodes within the same machine.

More than one node name can be specified. When multiple node names are specified, use a comma (,) to separate them.

Command input examples are given below. Each example assumes that the `prcls` command is executed at nodes `nd01` and `nd02`. Either of the following syntaxes may be used.

```
% delvcmd "prcls" -w nd01, nd02 <CR>
% delvcmd -w nd01, nd02 "prcls" <CR>
```

Before using this command, specify in it the full pathnames of `$DCDIR`, `$DCCONFPATH`, `$SHLIB_PATH`, or `$PATH` for each node.

Place either an apostrophe (') or quotation mark (") before and after any command to be specified as an argument. The following restrictions apply:

- Apostrophes (') should be used for MCF commands.
- Quotation marks (") should be used for TP1/Multi commands.

(2) Limitation on the value to be specified as command argument

An asterisk (*) cannot be used for the arguments of the `delvcmd` command. If an asterisk is used to specify multiple command arguments collectively, the `delvcmd` command may not execute normally.

Piping and redirecting are not allowed for a command name to be specified in the `delvcmd` command. However, the execution results of the `delvcmd` command can be piped or redirected. A command input example is given below.

```
% delvcmd "prcls" -w nd01, nd02 > file <CR>
```

(3) Commands which cannot be executed with the delvcmd command

Some commands cannot be executed with the `delvcmd` command due to the access permission set in the OpenTP1 system. In this case, the name of the user who executes the `delvcmd` command must be the same as that of the OpenTP1 administrator for the target node.

8.8 COBOL language templates

When writing a UAP in COBOL, the COBOL language templates can be used for easy coding of `DATA DIVISION`.

The COBOL language templates are stored in the `/BeTRAN/examples/COBOL/` directory.

8.8.1 Files of COBOL language templates

The COBOL language template is prepared for each OpenTP1 system service. The template file name is `DC`*xxx*`.cbl` (*xxx* is the last three characters of the COBOL-UAP creation program name.) The COBOL language template files are shown below.

`DCADM.cbl`: System operation management (CBLDCADM)

`DCDAM.cbl`: DAM file service (CBLDCDAM)

`DCDMB.cbl`: DAM file service (CBLDCDMB)

`DCIST.cbl`: IST service (CBLDCIST)

`DCJNL.cbl`: User journal output (CBLDCJNL)

`DCJUP.cbl`: User journal editing (CBLDCJUP)

`DCLCK.cbl`: Lock for resources (CBLDCLCK)

`DCLOG.cbl`: Message log output (CBLDCLOG)

`DCMCF.cbl`: Message exchange (CBLDCMCF)

`DCPRF.cbl`: Performance verification trace (CBLDCPRF)

`DCRAP.cbl`: Remote API facility (CBLDCRAP)

`DCRPC.cbl`: Remote procedure call (CBLDCRPC)

`DCRSV.cbl`: Remote procedure call (CBLDCRSV)

`DCTAM.cbl`: TAM file service (CBLDCTAM)

`DCTRN.cbl`: Transaction control (CBLDCTRN)

`DCUTO.cbl`: Online tester management (CBLDCUTO)

`DCXAT.cbl`: Association operating (CBLDCXAT)

8.8.2 How to use the cobol language templates

When using the COBOL language templates, modify the following values so that they can be suitable for processing of the UAP to be coded:

- Data area length (specific data only)

- Values substituted into each data area

For the values set into data area, see the syntax for each facility shown in the manual *OpenTP1 Programming Reference COBOL Language*.

There are two ways to use the COBOL language templates:

- Using the text editor calling facility
- Using the COPY statement of COBOL language

(1) How to use the text editor calling facility

To use a template:

1. Select a template for the desired facility from the `/BeTRAN/examples/COBOL/` directory.
2. Cut and paste the `DATA DIVISION` section to the source program of the UAP using the text editor calling facility.
3. Modify the pasted section so that it can be a data area suitable for the coding.

(2) How to use the COPY statement of COBOL language

To use a template:

1. Select a template for the desired facility from the `/BeTRAN/examples/COBOL/` directory.
2. Declare `COPY` with the file name of the template from the source program of the UAP. The file name to be specified with the `COPY` statement should be the file name of the template excluding the suffix `.cbl`.
3. Enter the file of the template in a directory which can be referenced with the `COPY` statement. This procedure depends on the processor of the COBOL language in use (such as file copy, setting environment variables).
4. Modify the file of the template so that it can be a data area suitable for the coding.

(3) Notes on using the COBOL language templates

1. The length of the `PICTURE` clause is declared as (*n*) in the data area to be modified according to the UAP processing. Modify the declaration before using it. Compilation without modification will result in error.
2. The following files of the COBOL language templates assume that the corresponding product has been installed:

   `DCDAM.cbl`, `DCDMB.cbl`: DAM file service (`CBLDCDAM`, `CBLDCDMB`)

   `DCTAM.cbl`: TAM file service (`CBLDCTAM`)

   `DCMCF.cbl`: Message exchange (`CBLDCMCF`)

`DCUTO.cbl`: Online tester management (`CBLDCUTO`)

`DCIST.cbl`: IST service (`CBLDCIST`)

3. The template for message exchange (`DCMCF.cbl`) contains all MCF-related information usable for OpenTP1. Therefore, some templates of the COBOL-UAP creation program cannot be used with some communication protocol supporting products. Values to be set in a data area also depend on the communication protocol supporting product. Change the format of `DCMCF.cbl` before using it, consulting the syntax of COBOL language shown in the applicable *OpenTP1 Protocol* manual.
4. It is recommended to copy a template from the original directory and then modify the copy according to the UAP processing.

8.9 How to use sample scenario template

OpenTP1 provides a scale-out scenario template. This sample enables you to use an OpenTP1 setup script file to be used in the scale-out scenario. To use this sample, you must have a JP1 product (JP1/AJS2, JP1/AJS2 - Scenario Operation, or JP1/Base) that is a prerequisite for JP1 scenario linkage. For details on the sample scenario template, see the description given in the manual *OpenTP1 Operation*.

8.10 How to use real-time acquisition item definition templates

OpenTP1 provides various real-time acquisition item definition files as templates that can be used with the real-time statistical information service. All of these files are contained in `/rts_template/examples/conf/` under the installation directory. To use a real-time acquisition item definition file, copy and place it immediately under `$DCCONFPATH/`.

The table below provides the file name and content of each real-time acquisition item definition file.

Table 8-11: File name and content of each real-time acquisition item definition file

File name	Contents
`base_itm`	Real-time acquisition item definition file for BASE
`dam_itm`	Real-time acquisition item definition file for DAM
`tam_itm`	Real-time acquisition item definition file for TAM
`all_itm`	Real-time acquisition item definition file for all statistical information
`none_itm`	Real-time acquisition item definition file for no statistical information
`mcfs_itm`	Real-time acquisition item definition file for the MCF (acquired for the entire system or for each server or service)
`mcfl_itm`	Real-time acquisition item definition file for the MCF (acquired for each logical terminal)
`mcfg_itm`	Real-time acquisition item definition file for the MCF (acquired for each service group)

For details on how to specify real-time acquisition item definitions, see the manual *OpenTP1 System Definition.*

Appendixes

A. Output Format of Undecided Transaction Information
B. Output Format of Deadlock Information
C. Examples of System Configurations Requiring Consideration of the Multi-Scheduler Facility

A. Output Format of Undecided Transaction Information

If `trn_tran_recovery_list = Y` is defined in the OpenTP1 transaction service definition at full recovery of OpenTP1, undecided transaction information can be output to the directory of the node of the transaction service.

(1) Names of directory and file to which undecided transaction information is output

The names of the directory and file to which undecided transaction information is output are as follows.

- Undecided transaction information is output to the directory `$DCDIR/spool/dctrninf/` of the node in which the transaction service exists.
- Every time full recovery of transaction service occurs, undecided transaction information is output as one file. The filename is `r1` + *transaction service starting time* (unique 8-digit hexadecimal number).

  This file name is displayed in the message log which indicates that undecided transaction information was output. Delete files which are no longer necessary.

Delete unnecessary files containing undecided transaction information following the procedure shown below:

- When deleting a file with a command:

  Execute the `trndlinf` command.
- When deleting information created previously in online mode at OpenTP1 activation:

  Specify the delete condition in the `trn_recovery_list_remove` and `trn_recovery_list_remove_level` operands of the transaction service definition.

(2) Output contents of undecided transaction information

The following items are output as undecided transaction information.

1. OpenTP1 system node ID

   System node ID of OpenTP1
2. Global transaction number

   Unique number for managing global transaction set by the system
3. Transaction branch number

   Unique number for managing transaction branch set by the system

4. Transaction's first status

   Processing status of transaction branch

5. Transaction's second status

   Status of transaction branch process

6. Transaction's third status

   Communication status of transaction branch

7. Process ID

   ID of the process operating the transaction branch

8. Server name

   Name of the server which started the transaction branch

9. Service name

   Name of the service which started the transaction branch

10. Transaction descriptor

    Index number to make distinction between transaction branches having the same transaction global ID

11. Branch descriptor

    Index number to make distinction between transaction branches that branched from one transaction branch. For root transaction branch, ********** is displayed.

12. Parent transaction descriptor

    Transaction identifier of the transaction which generated the corresponding transaction branch. For root transaction branch, ********** is displayed.

(3) Output format of undecided transaction information

Figure A-1 shows the output format of undecided transaction information. Figure A-2 gives an output example.

Figure A-1: Output format of undecided transaction information

```
                                                                                        (1)
Undecided transaction information                                    mmm dd HH:MM:SS yyyy

TRNGID            TRNBID            STATUS       PID         SERVER SERVICE ENTRYID    BRANCHID   PENTRYID
aaaaaaaabbbbbbbb  aaaaaaaacccccccc  dd..dd(e,f)  gggggggggg  hh..hh ii..ii  jjjjjjjjjj kkkkkkkkkk llllllllll   (2)
aaaaaaaabbbbbbbb  aaaaaaaacccccccc  dd..dd(e,f)  gggggggggg  hh..hh ii..ii  jjjjjjjjjj kkkkkkkkkk llllllllll
                                                           :
                                                           :
```

Explanation:

(1) Time at which full recovery was started

mmm: Month (lowercase letters)

dd: Day

HH: Hours

MM: Minutes

SS: Seconds

yyyy: Year (d, H, M, S, and y are digits.)

(2) Transaction information

aaaaaaaa:

OpenTP1 system node ID (8 characters)

bbbbbbbb:

Global transaction number (hexadecimal character string)

cccccccc:

Transaction branch number (hexadecimal character string)

dd...dd:

Transaction's first status (20 or less characters)

`BEGINNING`: Transaction branch start processing is underway.

`ACTIVE`: Executing

`SUSPENDED`: Suspended

`IDLE`: Changing to synchronization point processing

`PREPARE`: Under commit (phase 1) processing

`READY`: Waiting for commit (phase 2) processing

`HEURISTIC_COMMIT`: Heuristic decision commit processing is underway.

`HEURISTIC_ROLLBACK`: Heuristic decision rollback processing is underway.

`COMMIT`: Commit processing is underway.

`ROLLBACK_ACTIVE`: Waiting for rollback processing

`ROLLBACK`: Rollback processing is underway.

`HEURISTIC_FORGETTING`: Transaction branch termination processing after heuristic decision is underway.

`FORGETTING`: Transaction branch termination processing is underway

e:

Transaction's second status (1 character)

`u`: User server executing a user server process

`r`: Transaction branch recovery processing in a transaction recovery process

`p`: Waiting for completion of recovery other transaction branch processing in a transaction recovery process

When the first state is READY and the root transaction branch is not in the same node, direction by the user is awaited.

f:

Transaction's third status (1 character)

s: In sending

r: In receiving

n: Not in sending or receiving

"In sending or receiving" means that the transaction manager is in progress of communication for synchronization between the transaction branches.

gg...gg:

Process ID (decimal number)

hh...hh:

Server name (8 or less characters)

ii...ii:

Service name (32 or less characters) (For SUP, spaces are set.)

jjjjjjjjj:

Transaction identifier (decimal number)

kkkkkkkkk:

Branch identifier (decimal number)#

lllllllll:

Originating transaction identifier (decimal number)#

#

For root transaction branch, `**********` is displayed.

Figure A-2: Output example of undecided transaction information

```
Undecided transaction information                                    Jun 10 12:43:55 1996

TRNGID           TRNBID           STATUS             PID SERVER^SERVICE ENTRYID   BRANCHID   PENTRYID
@@@@hst100000001 @@@@hst100000001 COMMIT(p,n)          0 sup1                 0 ********** **********
@@@@hst100000001 @@@@hst100000006 COMMIT(p,n)          0 spp1   sv1           1          0          0
@@@@hst100000002 @@@@hst10000000a ROLLBACK(p,n)        0 sup2                 2 ********** **********
@@@@hst100000002 @@@@hst10000000e ROLLBACK(p,n)        0 spp2   sv1           3         32          2
                                                       :
                                                       :
```

B. Output Format of Deadlock Information

Assume that a deadlock occurs between two or more UAPs. In this case, if `lck_deadlock_info = Y` is defined in the OpenTP1 lock service definition, deadlock information is output to the directory in the node of the lock service. This information is output in the following cases:

- The lock service detects a deadlock (deadlock information)
- A timeout occurs when waiting for lock to be released (timeout information)

(1) Names of directory and file to which deadlock information is output

Deadlock information is output as follows.

- Deadlock information is output to the directory `$DCDIR/spool/dclckinf/` of the node containing the lock service which detected the deadlock or timeout.
- Every time deadlock information occurs, it is output as one file. The date and time at which the deadlock or timeout occurred are used as the file name. The file name length differs depending on whether the date is one or two digits.

Example

```
Oct. 3, 7 h. 41 m. 00 s. : Oct3074100
Oct. 10, 15 h. 5 m. 27 s. : Oct10150527
```

This file name is displayed in the message log which indicates that a deadlock occurred. Delete files which are no longer necessary.

(2) Output format of deadlock information

Figure B-1 shows the output format of deadlock information displayed when a deadlock is detected. Figure B-2 gives an output example.

Figure B-1: Output format of deadlock information

```
                                                                          (1)
    Deadlock  information                                      Jul DD HH:MM:SS YYYY

(2)  server:CCCCCCCC    pid:CCCCC
(3)  GID:CCCCCCCCCCCCCCCC
(4)  BID:CCCCCCCCCCCCCCCC
     occupy
(5)    server name:CCCCCCCC  mode:CC  resource name:CCCCCCCCCCCCCCCC  owner:CCCCCCC
     wait
(6)    server name:CCCCCCCC  mode:CC  resource name:CCCCCCCCCCCCCCCC  owner:CCCCCCC
(7)   wait start time HH:MM:SS

    server:CCCCCCCC    pid:CCCCC
    GID:CCCCCCCCCCCCCCCC
    BID:CCCCCCCCCCCCCCCC
     occupy
      server name:CCCCCCCC  mode:CC  resource name:CCCCCCCCCCCCCCCC  owner:CCCCCCC
     wait
      server name:CCCCCCCC  mode:CC  resource name:CCCCCCCCCCCCCCCC  owner:CCCCCCC
     wait start time HH:MM:SS
                                      :
                                      :
                                      :
```

Explanation:

(1) Time at which the deadlock was detected

(2) Server name and process ID of the access requester

(3) Transaction global identifier of the access requester

(4) Transaction branch identifier of the access requester

(5) Information on the server acquiring the resources

- Name of the server requesting lock
- Lock mode (PR or EX)
- Name of the occupied resource
- MIGRATE/BRANCH request type

(6) Information on the server waiting for resource unlocking

- Name of the server requesting lock
- Lock mode (PR or EX)
- Name of the resource to be unlocked

- MIGRATE/BRANCH request type

(7) Time at which resource unlock wait occurred

Note

Items (2) to (7) are output for each UAP (server process) involved in the deadlock.

Figure B-2: Output example of deadlock information

```
    Deadlock  information                                          Jan 27 14:41:43 1995

(2)  server:spp2        pid:2076
(3)  GID:@@@@nd0100000001
(4)  BID:@@@@nd0100000002
     occupy
(5)    server name:usr       mode:EX  resource name:file2            owner:BRANCH
     wait
(6)    server name:          mode:    resource name:                 owner:
(7)   wait start time  :  :

    server:sup2        pid:2078
    GID:@@@@nd0100000001
    BID:@@@@nd0100000001
     occupy
      server name:          mode:    resource name:                 owner:       ] #
     wait
      server name:usr       mode:EX  resource name:file1            owner:BRANCH
     wait start time 14:41:40

    server:sup3        pid:2079
    GID:@@@@nd0100000012
    BID:@@@@nd0100000020
     occupy
      server name:usr       mode:EX  resource name:file1            owner:BRANCH
     wait
      server name:usr       mode:EX  resource name:file2            owner:BRANCH
     wait start time 14:41:27
```

#

When there is no occupied resource, this field is displayed as blank.

(3) Output format of timeout information

Figure B-3 shows the output format of timeout information displayed when a timeout is detected. Figure B-4 gives an output example.

Figure B-3: Output format of timeout information

```
                                                                 (1)
   Timeout  information                                    Jul DD HH:MM:SS YYYY

(2)  server:CCCCCCCC  pid:CCCCC
(3)  GID:CCCCCCCCCCCCCCCC
(4)  BID:CCCCCCCCCCCCCCCC
     occupy
(5)   server name:CCCCCCCC  mode:CC  resource name:CCCCCCCCCCCCCCCC  owner:CCCCCCC
      server name:CCCCCCCC  mode:CC  resource name:CCCCCCCCCCCCCCCC  owner:CCCCCCC
                                  :
                                  :
     wait
(6)    server name:CCCCCCCC  mode:CC  resource name:CCCCCCCCCCCCCCCC  owner:CCCCCCC

(7)  server:CCCCCCCC   pid:CCCCC
(8)  GID:CCCCCCCCCCCCCCCC
(9)  BID:CCCCCCCCCCCCCCCC
     occupy
(10)    server name:CCCCCCCC  mode:CC  resource name:CCCCCCCCCCCCCCCC  owner:CCCCCCC

    server:CCCCCCCC  pid:CCCCC
    GID:CCCCCCCCCCCCCCCC
    BID:CCCCCCCCCCCCCCCC
     occupy
      server name:CCCCCCCC  mode:CC  resource name:CCCCCCCCCCCCCCCC  owner:CCCCCCC

    server:CCCCCCCC  pid:CCCCC
    GID:CCCCCCCCCCCCCCCC
    BID:CCCCCCCCCCCCCCCC
     occupy
      server name:CCCCCCCC  mode:CC  resource name:CCCCCCCCCCCCCCCC  owner:CCCCCCC

    server:CCCCCCCC  pid:CCCCC
    GID:CCCCCCCCCCCCCCCC
    BID:CCCCCCCCCCCCCCCC
     occupy
      server name:CCCCCCCC  mode:CC  resource name:CCCCCCCCCCCCCCCC  owner:CCCCCCC
                                 :
                                 :
                                 :
```

Explanation:

(1) Time at which the timeout was detected

(2) Server name and process ID related to the timeout

(3) Transaction global identifier related to the timeout

(4) Transaction branch identifier related to the timeout

(5) Information that was exclusively used by the server having caused the timeout.

This information is output for all resources that were occupied by the server having caused the timeout.

- Name of the server requesting lock
- Lock mode (PR or EX)
- Name of the occupied resource
- MIGRATE/BRANCH request type

(6) Wait information on the server having caused the timeout

- Name of the server requesting lock
- Lock mode (PR or EX)
- Name of the resource to be unlocked
- MIGRATE/BRANCH request type

(7) Name and process ID of the server constituting the factors of the timeout

(8) Transaction global identifier constituting the factors of the timeout

(9) Transaction branch identifier constituting the factors of the timeout

(10) Information that was exclusively used by the server constituting the factors of the timeout. This information is output for all servers constituting the factors of the timeout.

- Name of the server requesting lock
- Lock mode (PR or EX)
- Name of the occupied resource
- MIGRATE/BRANCH request type

Note

Items (7) to (10) (concerning a server waiting for resource acquisition or unlocking) are output for each resource which was about to be occupied by the server having caused the timeout when the timeout occurred.

Figure B-4: Output example of timeout information

```
Timeout  information                                        Jun 30 16:25:45 1995


server:sup1        pid:2940
GID:@@@@nd0100000008
BID:@@@@nd0100000008
 occupy
  server name:usr       mode:PR  resource name:fileC            owner:BRANCH    ]
  server name:usr       mode:EX  resource name:fileB            owner:BRANCH    ] #
  server name:usr       mode:PR  resource name:fileE            owner:BRANCH    ]
 wait
  server name:usr       mode:EX  resource name:fileA            owner:BRANCH

server:sup3        pid:2944
GID:@@@@nd0100000003
BID:@@@@nd0100000004
 occupy
  server name:usr       mode:PR  resource name:fileA            owner:BRANCH

server:sup2        pid:2941
GID:@@@@nd0100000004
BID:@@@@nd0100000005
 occupy
  server name:usr       mode:PR  resource name:fileA            owner:BRANCH
```

#

When there are no occupied resources, this information is not output.

(4) Output format used with TP1/FS/Table Access

If the TP1/FS/Table Access is in use and use of its resource encounters deadlock time-out, the output information will contain the table name, key values, among others.

The figure below shows the format of information which will be output when a deadlock is detected.

Figure B-5: Output format of TAM resource deadlock information

```
      Deadlock information                                        Dec 15 17:26:30 1995
       server:sups01x     pid:7786
       GID:b81eBROS00000004
       BID:BROSBROS00000004
        occupy
         server name:_tam      mode:EX  resource name:R000010000000523  owner:MIGRATE
(1)                  --> TAM Table name = [tam_primary_table]
(2)                  --> TAM Record Key (Length=[10])
(3)               [ 30 30 30 30 30 30 30 35 32 32 00 00 00 00 00 00 ]:0000000522......
        wait
         server name:_tam      mode:EX  resource name:R000010000000522  owner:MIGRATE
                      --> TAM Table name = [tam_primary_table]
                      --> TAM Record Key (Length=[10])
                   [ 30 30 30 30 30 30 30 35 32 31 00 00 00 00 00 00 ]:0000000521......
        wait start time 17:26:28
       server:sups01t     pid:7781
       GID:b81dBROS00000003
       BID:BROSBROS00000003
        occupy
         server name:_tam      mode:EX  resource name:T00003            owner:MIGRATE
                     --> TAM Table name = [tam_primary_table]
                     --> TAM Record Key (Length=[10])
                  [ 30 30 30 30 30 30 30 35 32 31 00 00 00 00 00 00 ]:0000000521......
        wait
         server name:_tam      mode:EX  resource name:T00001            owner:MIGRATE
                     --> TAM Table name = [tam_primary_table]
                     --> TAM Record Key (Length=[10])
                  [ 30 30 30 30 30 30 30 35 32 32 00 00 00 00 00 00 ]:0000000522......
        wait start time 17:26:28
```

Explanation:

(1) TAM table name indicated by the resource name

If the resource name begins with `T`, the TAM table indicated by (1) is the resource; lines (2) and (3) are not output.

If the resource name begins with `R`, the TAM record is the resource; it is on the TAM table indicated by (1). In this case, lines (2) and (3) are output.

(2) Key length of the TAM record indicated by the resource name

(3) Key value of the TAM record indicated by the resource name

The key value is given in [ ] in hexadecimal.

If the key value is printable, the printable characters are output to the right of [ ]; if it is not, `'.'` is output to the right of [ ]. If the key value is less than a multiple of 16, the remaining area is padded with `00`s.

C. Examples of System Configurations Requiring Consideration of the Multi-Scheduler Facility

As systems become larger and machines and networks boast increasingly better performances, conventional schedulers may experience difficulty scheduling messages efficiently. This appendix gives examples of system configurations for which you should consider the multi-scheduler facility and examples of resolutions.

This appendix contains the following sections:

C.1 Overview of processing by the scheduler facility

C.2 Examples of system configurations in which the scheduler is likely to be the cause of error

C.3 Example of a system configuration using the multi-scheduler facility

C.4 Notes

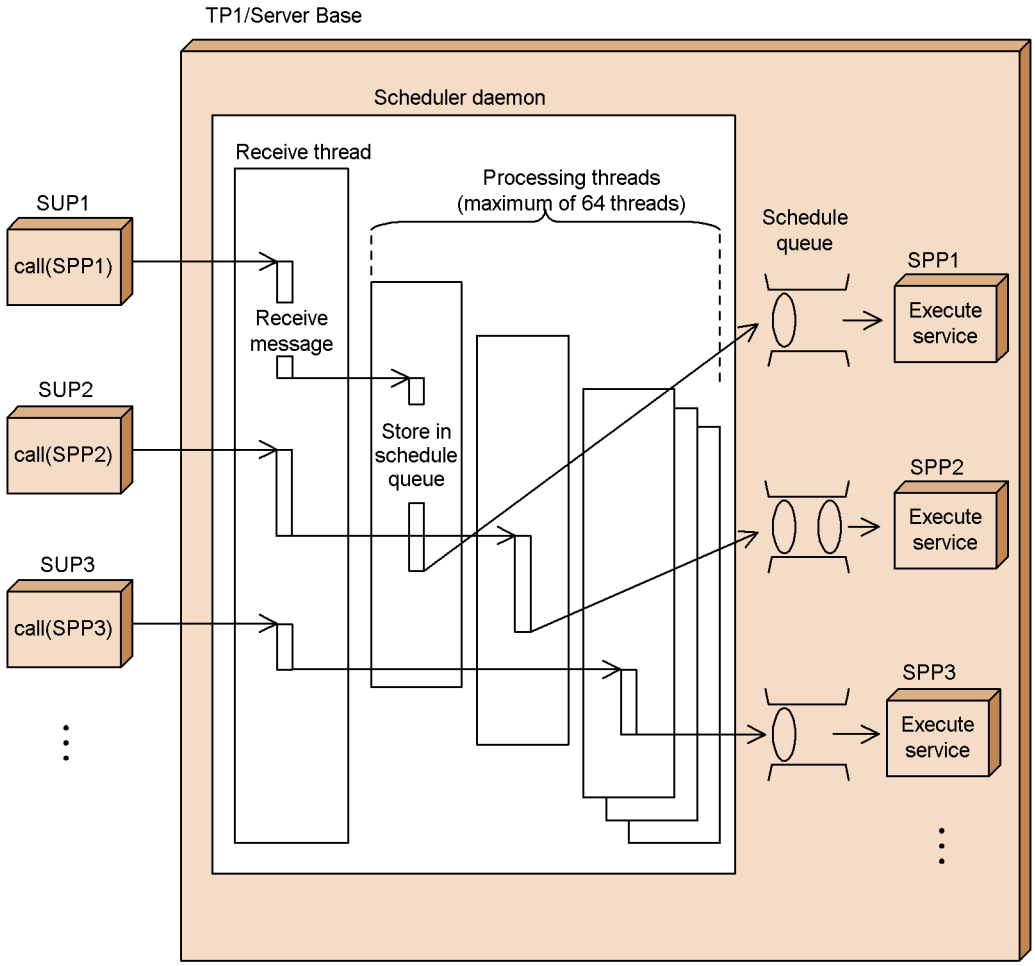
C.1 Overview of processing by the scheduler facility

When a client UAP requests a service from a queue-receiving server (SPP that uses a schedule) on a remote node, the scheduler daemon of the node that contains the requested server receives the service request message and stores it in the schedule queue on the queue-receiving server.

The scheduler daemon consists of a receive thread (one thread) which receives service request messages from client UAPs and processing threads (maximum of 64 threads) which store service requests in a schedule.

The figure below shows an overview of processing by the scheduler facility.

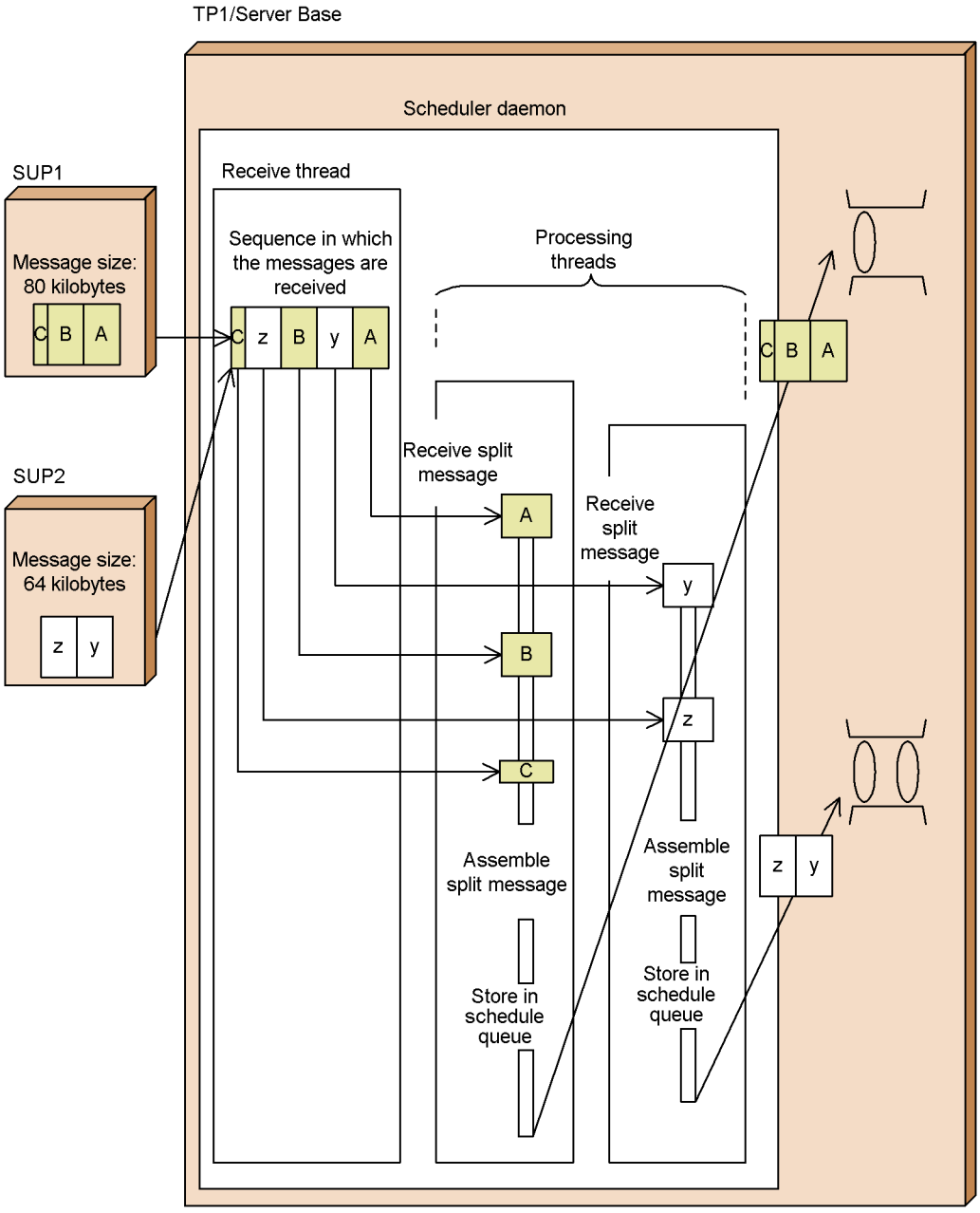
Figure C-1: Overview of processing by the scheduler facility



The receive thread of the scheduler daemon will receive up to 32 kilobytes of a service request message at one time from a client UAP. If a service request message exceeds 32 kilobytes, the system splits the message when it is sent and received. This prevents the scheduler daemon from being exclusively occupied for receiving a message from a single client UAP.

The figure below shows an overview of processing service request messages.

Figure C-2: Overview of processing service request messages



As illustrated in Figure C-2, the scheduler schedules service request messages received

from client UAPs.

C.2 Examples of system configurations in which the scheduler is likely to be the cause of error

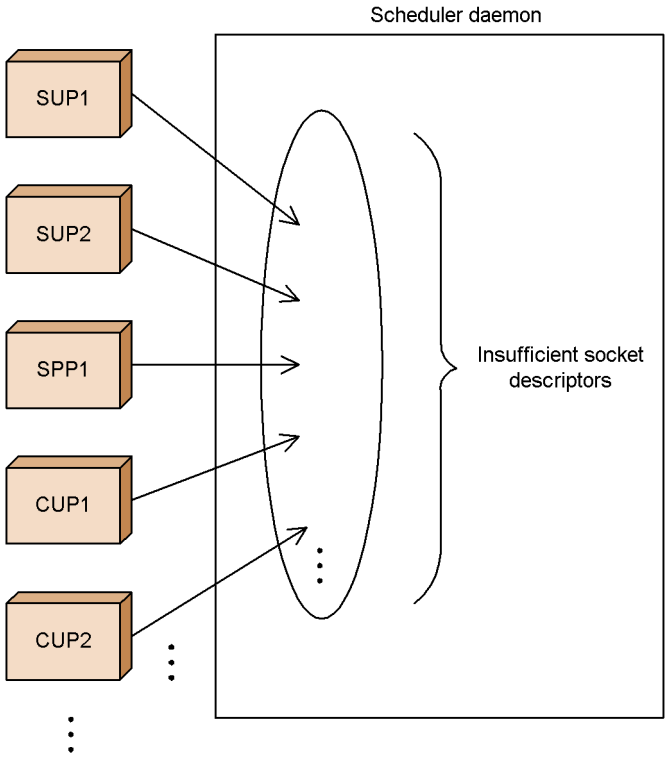
As systems become larger and machines and networks boast increasingly better performances, conventional scheduler daemons may experience difficulty scheduling messages efficiently. This section gives examples of system configurations in which the scheduler is likely to be the cause of error.

(1) System with insufficient socket descriptors

When the number of client UAPs to be connected to a single scheduler daemon increases, you may not be able to specify a sufficient number of socket descriptors to be used by the scheduler daemon. If there are insufficient socket descriptors for the scheduler daemon, the system requests disconnection and then ends a connection in order to reserve new socket descriptors. Depending on the load exerted on the system by this disconnection processing, the scheduling performance of the scheduler daemon may drop.

The figure below shows an example of a system that has insufficient socket descriptors.

Figure C-3: Example of a system that has insufficient socket descriptors



(2) System in which the connect system call encounters an error

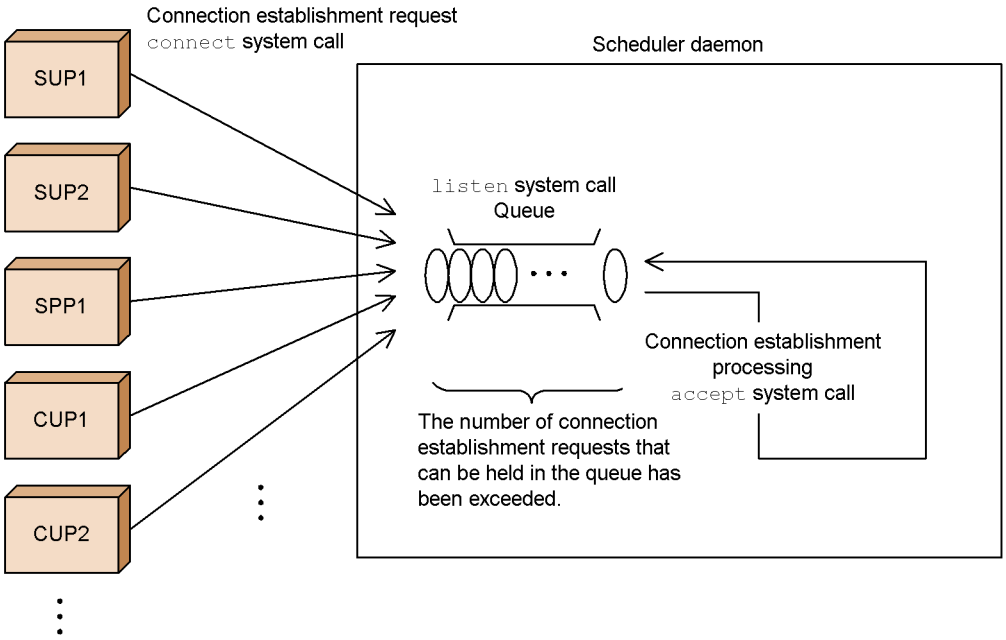
OpenTP1 uses TCP/IP as its communication protocol. Therefore, a connection establishment request (`connect` system call) from a client UAP is held in a wait queue of the `listen` system call until it is fetched by the accept system call.

The number of connection establishment requests that can be held in the wait queue depends on the operating system. However, if client UAPs send numerous requests at one time, the number of generated connection establishment requests may exceed the number of requests that can be held in the queue.

If client UAPs generate more connection establishment requests than can be held in the wait queue, CUP (TP1/Client) outputs the message `KFCA02449-E`, and SUP and SPP (TP1/Server Base) output the message `KFCA00327-W`. The system may consider that the service requests failed due to a communication error or because OpenTP1 was not started.

The figure below shows an example of a system in which the `connect` system call encountered an error.

Figure C-4: Example of a system in which the connect system call encountered an error



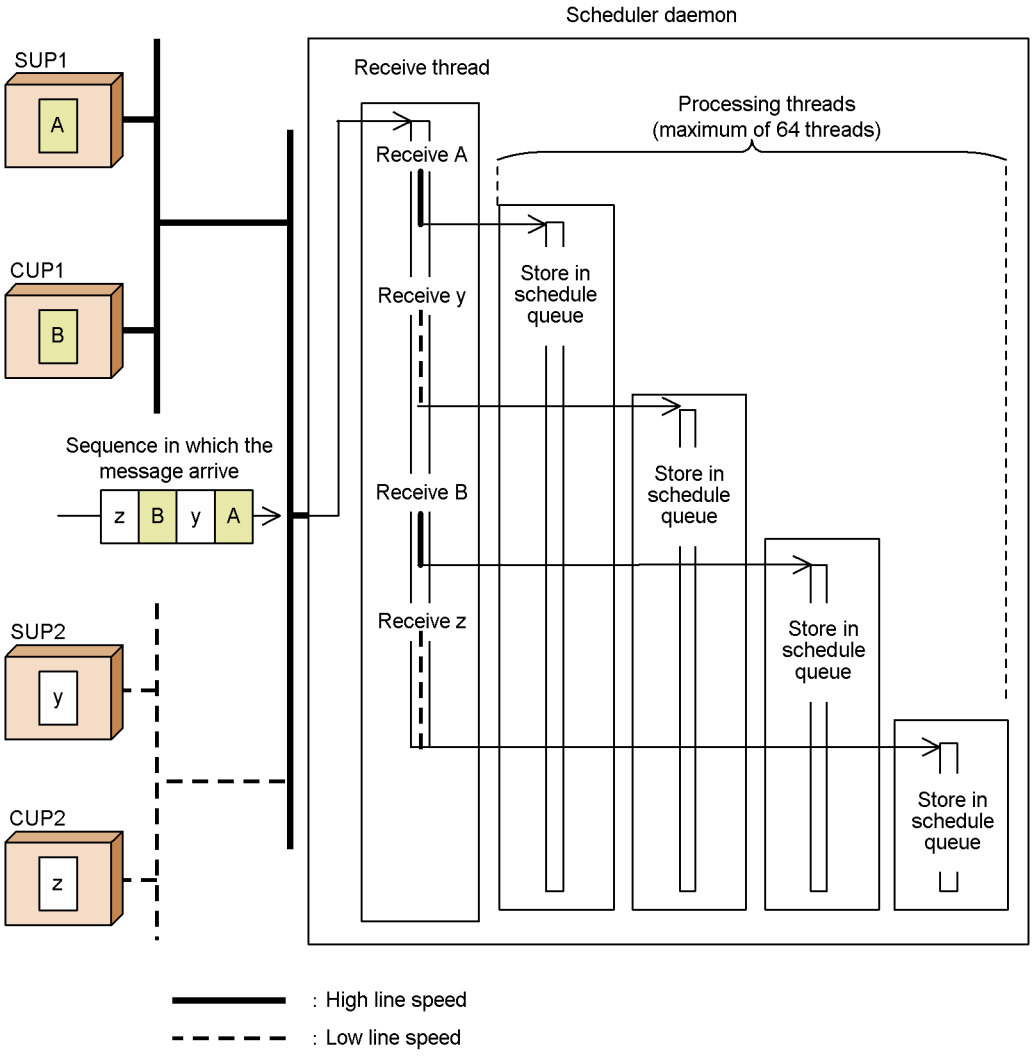
(3) System using networks that have different line speeds

While the receive thread of the scheduler daemon is receiving a service request message from a particular client UAP, it cannot receive a service request message from another client UAP. (However, if a service request message exceeds 32 kilobytes, it is split before being sent.)

Therefore, processing for receiving a message from a client UAP connected to a network that has a low line speed delays processing for receiving a message from a client UAP connected to a network that has a high line speed. The performance of the network that has a high line speed may thus be compromised.

The figure below shows an example of a system that uses networks with different line speeds. This example compares the two lines having different speeds. A comparison of lines where one line is twice as fast as the other line shows that receive processing for the slow line takes twice as long as that for the fast line. This difference in the processing time corresponds exactly to the difference in the line speed.

Figure C-5: Example of a system using networks that have different line speeds

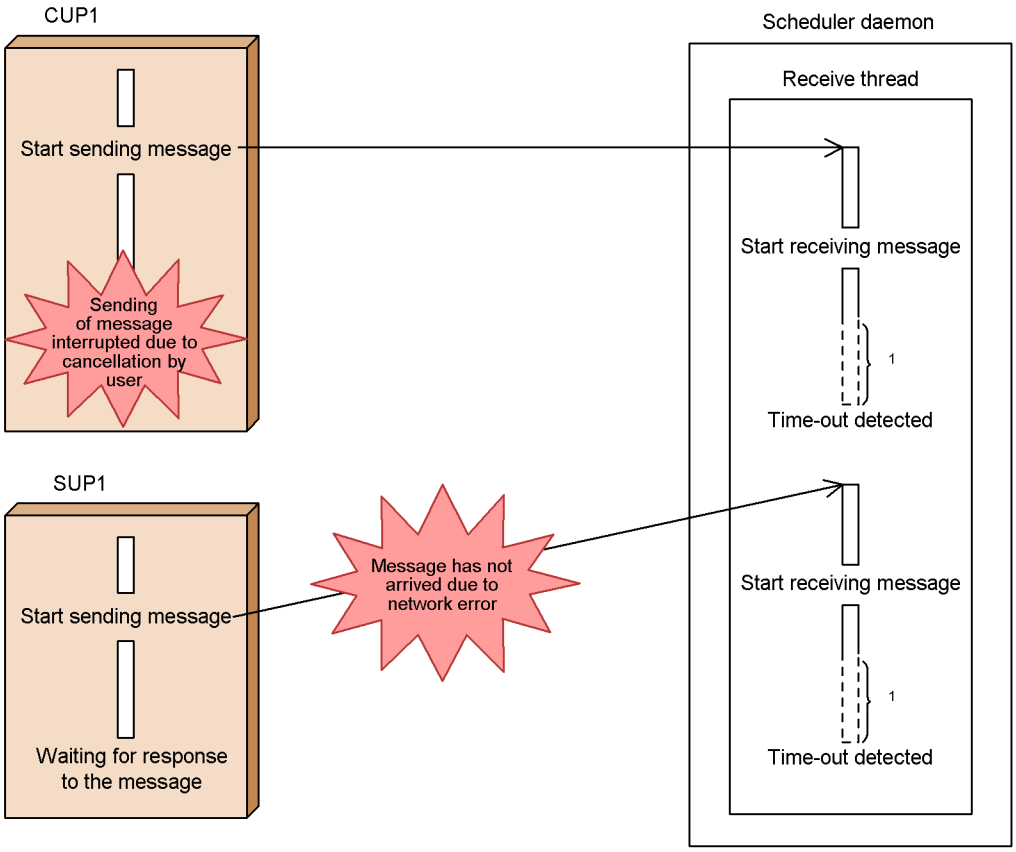


(4) System in which service request messages are interrupted

Assume that processing for sending a service request message to the scheduler daemon is interrupted because, for instance, the client UAP was forcibly terminated. In this case, scheduling may be delayed until message receive processing by the receive thread reaches a time-out.

The figure below shows an example of a system in which service request messages are interrupted.

Figure C-6: Example of a system in which service request messages are interrupted



[1] Receive processing is suspended before time-out is detected.

(5) System in which the processing threads are temporarily deficient

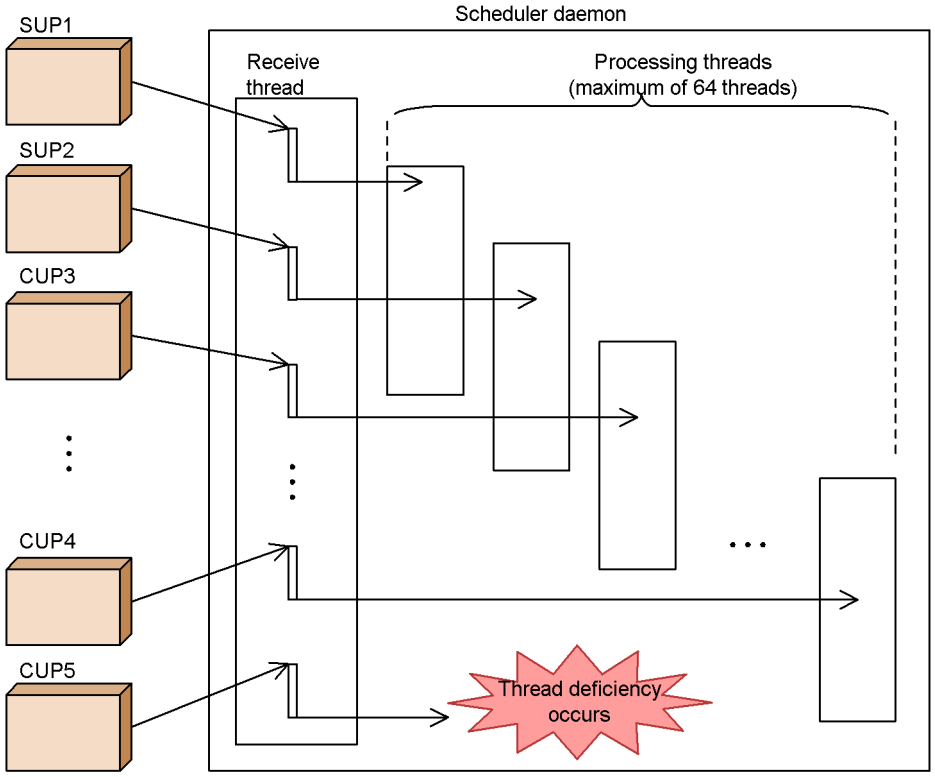
If the scheduler daemon receives a very large number of service request messages at one time from client UAPs and does not have the capacity to process them, there may be a temporary deficiency of processing threads.

If a deficiency of processing threads occurs, the system outputs the message `KFCA00356-W` and may temporarily consider that the service requests failed due to a communication error or time-out.

Specify the timing for outputting the message `KFCA00356-W` in `rpc_server_busy_count` in the system common definition. For details, see the manual *OpenTP1 System Definition*.

The figure below shows an example of a system in which the processing threads are temporarily deficient.

Figure C-7: Example of a system in which the processing threads are temporarily deficient



C.3 Example of a system configuration using the multi-scheduler facility

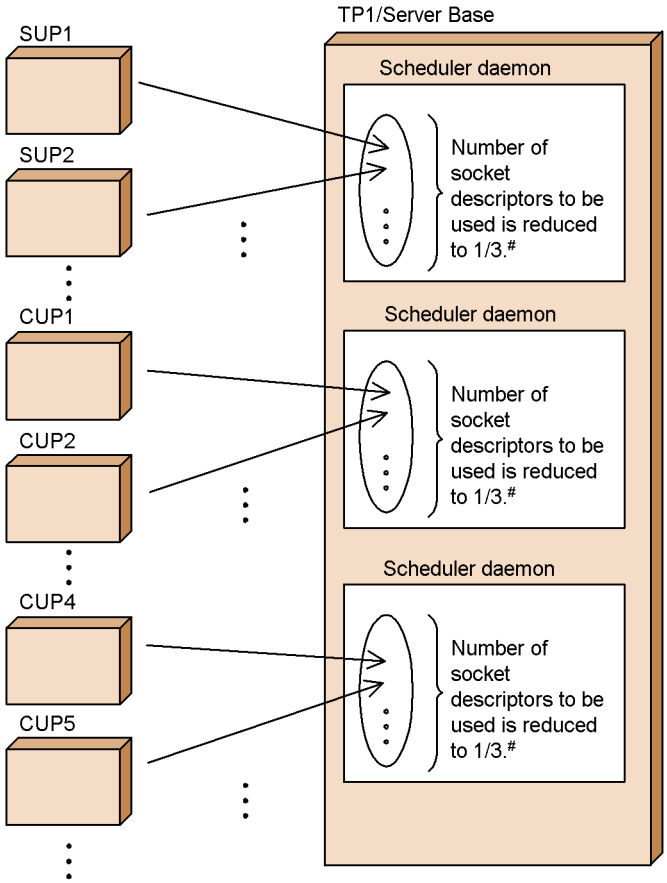
By using the multi-scheduler facility, you can resolve problems that are likely to be caused by the scheduler. The following describes system configurations that use the multi-scheduler facility.

(1) System configuration for solving the deficiency of socket descriptors

You can reduce the number of socket descriptors to be used by a single scheduler daemon by distributing the client UAPs to be connected to scheduler daemons.

The figure below shows an example of a system configuration that solves the deficiency of socket descriptors.

Figure C-8: Example of a system configuration that solves the deficiency of socket descriptors



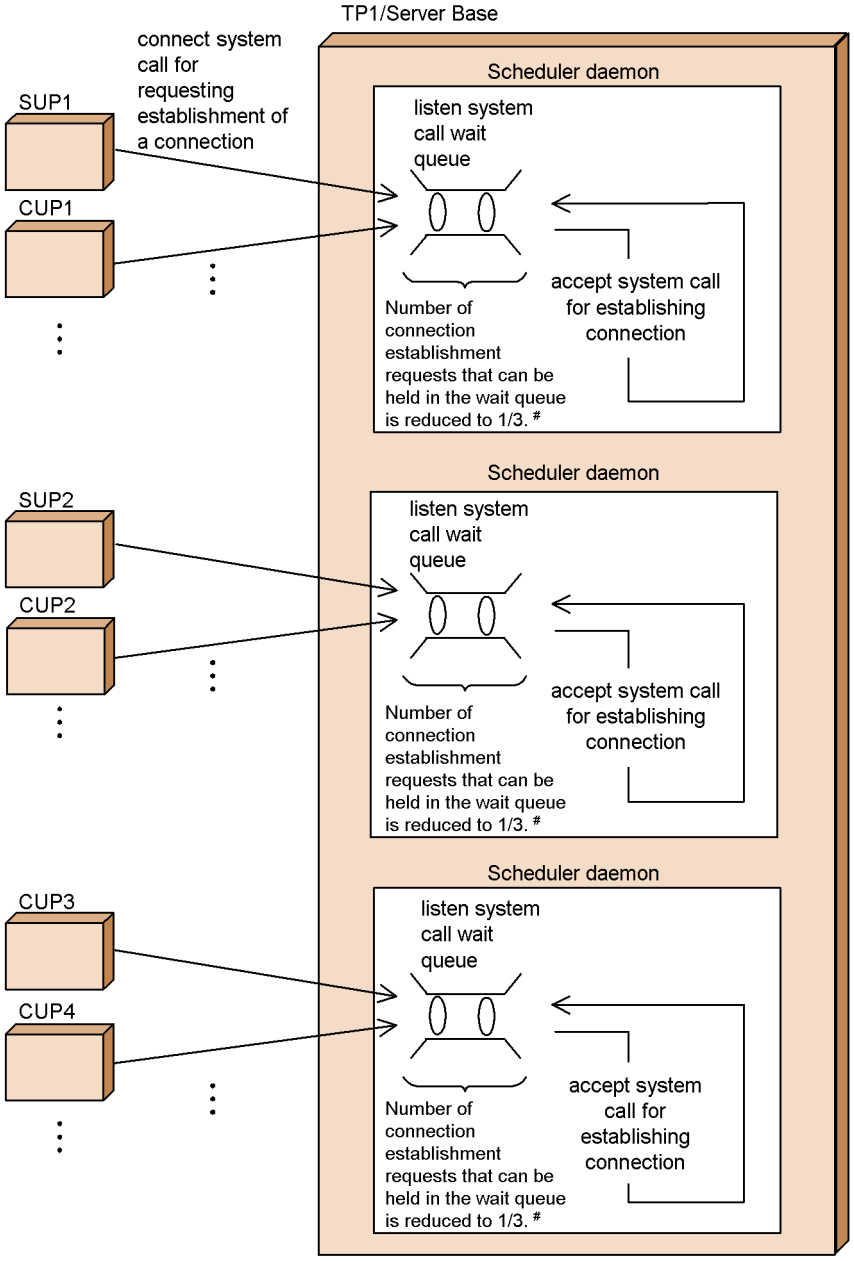
When you specify 3 as the number of scheduler daemons.

(2) System configuration for solving errors with the connect system call

You can reduce the number of connection establishment requests to be held in the wait queue for the listen system call by distributing the client UAPs to be connected to scheduler daemons.

The figure below shows an example of a system configuration that solves errors with the connect system call.

Figure C-9: Example of a system configuration that solves errors with the connect system call



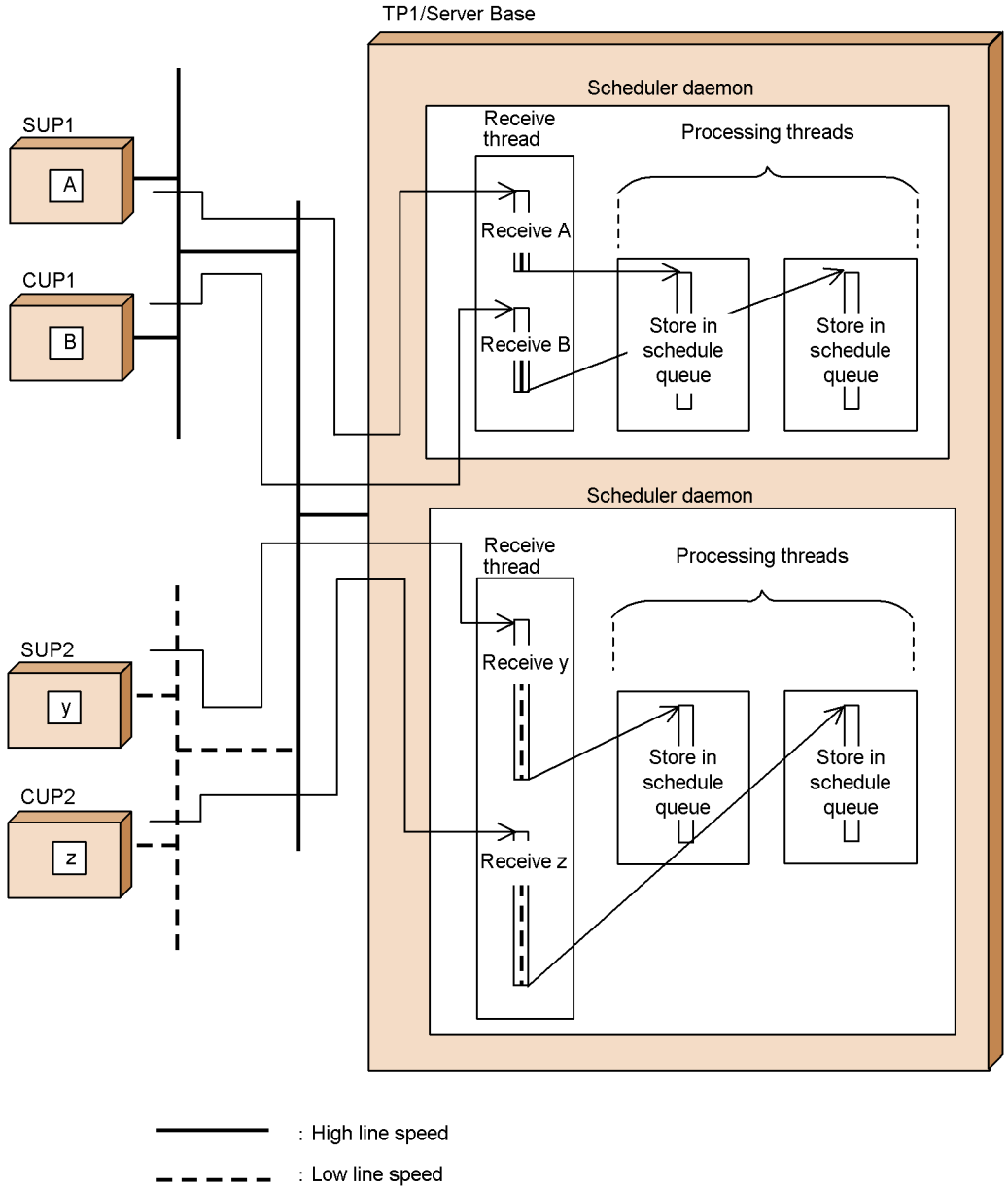
When you specify 3 as the number of scheduler daemons.

(3) System that can effectively use a network having a high line speed (when the system uses networks having different line speeds)

You can effectively use a network having a high line speed by separating the scheduler daemon for processing client UAPs on a network having a high line speed from the scheduler daemon for processing client UAPs on a network having a low line speed.

The figure below shows an example of a system that effectively uses a network having a high line speed.

Figure C-10: Example of a system that effectively uses a network having a high line speed

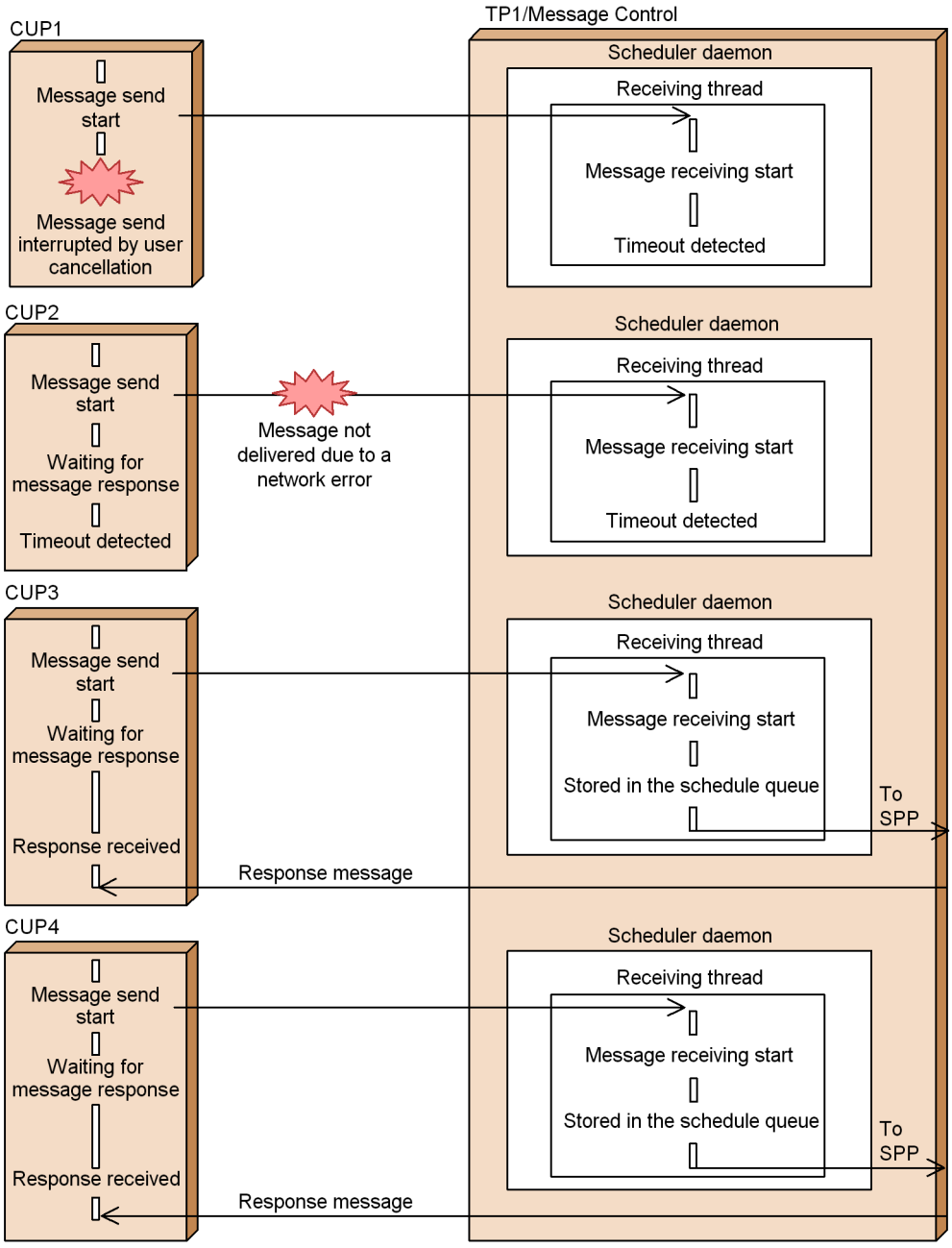


(4) System in which service request messages are not interrupted

By using the multi-scheduler facility and localizing scheduler daemons whose message receive processing is interrupted, you can schedule messages without delaying other service request messages.

The figure below shows an example of a system in which service request messages are not interrupted.

Figure C-11: Example of a system in which service request messages are not interrupted

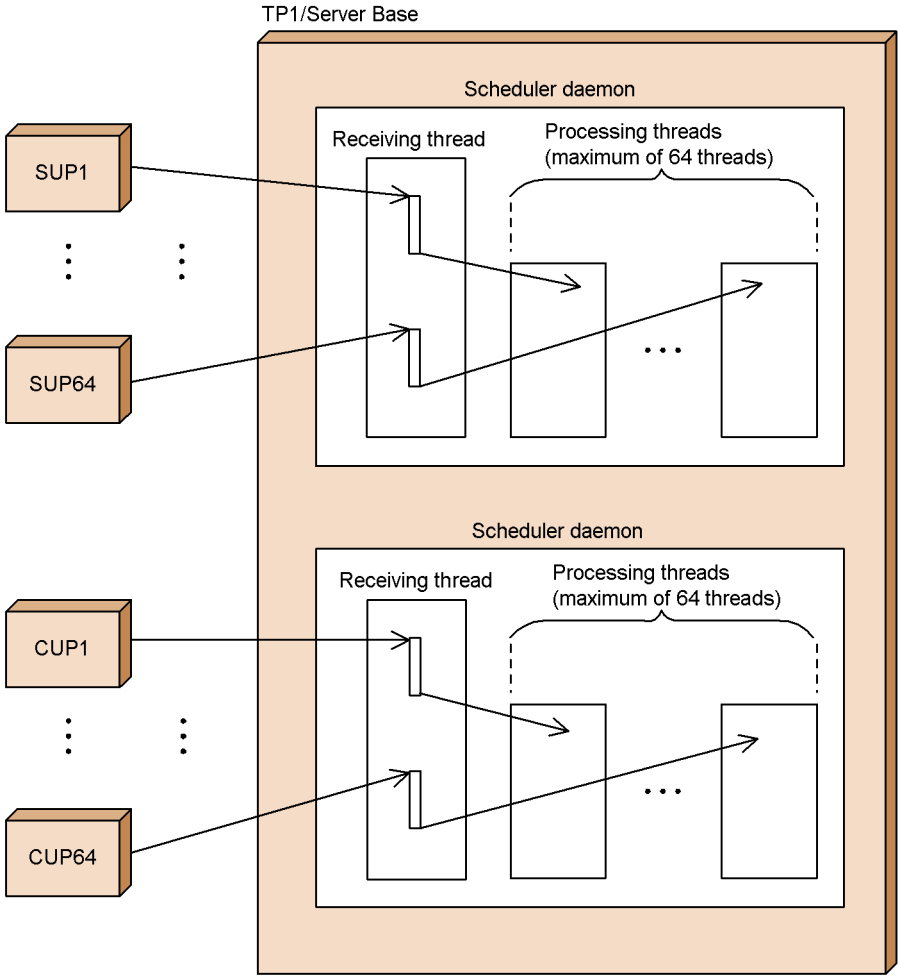


(5) System with an increased number of simultaneously executable processing threads

You can use the multi-scheduler facility to increase the number of processing threads that can be executed at one time. This prevents temporary communication errors and time-out caused by a deficiency of processing threads.

The figure below shows an example of a system with an increased number of simultaneously executable processing threads.

Figure C-12: Example of a system with an increased number of simultaneously executable processing threads



C.4 Notes

1. Installation of TP1/Extension 1 is a prerequisite for using the multi-scheduler facility. If you have not installed TP1/Extension 1, operation is not guaranteed. For details on the multi-scheduler facility, see the manual *OpenTP1 System Definition* or *OpenTP1 TP1/Client User's Guide TP1/Client/W, TP1/Client/P*.
2. When applying the multi-scheduler facility, you have to change the system definitions. You may also have to change the kernel parameters if you increase the number of scheduler daemons.
3. If a service group within the OpenTP1 system contains both a user server that uses the multi-scheduler facility and a user server that does not, note the following:
   - All service requests issued using the multi-scheduler facility will initially be allocated to the user server that uses the multi-scheduler facility for load distribution.
   - Service requests issued using the multi-scheduler facility will not be allocated to the unequipped user server, even in situations when the multi-scheduler facility equipped user server is heavily loaded. To distribute the load from a heavily-loaded user server equipped with the multi-scheduler facility to an unequipped user server, specify the `-t` option to the `scdmulti` definition command for schedule service definition. For details on the `scdmulti` definition command, see the description of the schedule service definition in the manual *OpenTP1 System Definition*.

Index

Symbols

$DCDIR/aplib/ directory 36

A

abbreviations for products iv
accessing
    from transaction process to DAM file block 291
    notes on, from UAP 287
    TAM table 314
    to DAM file in offline mode 297
    to DAM file in online mode 288
    to DAM files 287
    to unrecoverable DAM file 303
acquiring
    node address of client UAP 83
    OpenTP1 node identifier 413
    OpenTP1 node status 410
    performance verification trace 186
    synchronization point 122
    TAM table information 319
    TAM table status 318
    user server status 411
acronyms ix
ans 209
ANSI C 28
application attribute 256
    MHP for MCF event 255
application name 19, 37, 216
application program 1, 2, 8
    creating 26
    setting up environment 36
    starting 233
application startup process 255, 279, 281
application timer start requests, deleting 199
asynchronous message reception 216
asynchronous message send processing 217
asynchronous prepare optimization 141
asynchronous-response-type RPC 76
    receiving response of 76
    rejection of receiving processing results 80
    time monitoring of 77
atomic_update 127
audit log, outputting 173
auto_restart 11
automatic connection mode 116
automatic startup 73

B

balance_count 38, 90
Base sample 420, 428
basic OpenTP1 facility 71
bind mode 35, 36
bind mode for linkage 35, 36
block 286
block-based lock 302
blocking timeout 378, 382
branch send mode 208
buffer format 1 215
buffer format 2 215

C

C language 28
C or C++ 28
C++ language 28
CBLDCADM('COMMAND ') 155
CBLDCADM('COMPLETE') 12, 164
CBLDCADM('STATUS ') 165
CBLDCDAM('CLOS') 288
CBLDCDAM('END ') 304
CBLDCDAM('HOLD') 289
CBLDCDAM('OPEN') 288
CBLDCDAM('READ') 289
CBLDCDAM('REWT') 289
CBLDCDAM('RLES') 289
CBLDCDAM('STAT') 296

CBLDCDAM('STRT') 304
CBLDCDAM('WRIT') 289
CBLDCDMB('BSEK') 298
CBLDCDMB('CLOSE') 297
CBLDCDMB('CRAT') 300
CBLDCDMB('DGET') 298
CBLDCDMB('DPUT') 298
CBLDCDMB('GET ') 297
CBLDCDMB('OPEN') 297
CBLDCDMB('PUT ') 298
CBLDCIST('CLOS') 355
CBLDCIST('OPEN') 354
CBLDCIST('READ') 354
CBLDCIST('WRIT') 354
CBLDCJNL('UJPUT ') 176
CBLDCJUP('CLOSERPT') 178
CBLDCJUP('OPENRPT') 178
CBLDCJUP('RDGETRPT') 178
CBLDCLCK('GET') 361
CBLDCLCK('RELALL ') 362
CBLDCLCK('RELNAME ') 362
CBLDCLOG('PRINT ') 169
CBLDCMCF('ADLTAP ') 199
CBLDCMCF('APINFO ') 51
CBLDCMCF('CLOSE ') 18, 23
CBLDCMCF('COMMIT ') 228
CBLDCMCF('CONTEND ') 207, 223
CBLDCMCF('EXECAP ') 233
CBLDCMCF('MAINLOOP') 23, 32
CBLDCMCF('OPEN ') 18, 23
CBLDCMCF('RECEIVE ') 207, 216
CBLDCMCF('RECVSYNC') 208, 219
CBLDCMCF('REPLY ') 207, 217
CBLDCMCF('RESEND ') 226
CBLDCMCF('ROLLBACK') 229
CBLDCMCF('SEND ') 208, 217
CBLDCMCF('SENDRECV') 208, 219
CBLDCMCF('SENDSYNC') 208, 219
CBLDCMCF('TACTCN ') 191
CBLDCMCF('TACTLE ') 200
CBLDCMCF('TDCTCN ') 191
CBLDCMCF('TDCTLE ') 200
CBLDCMCF('TDLQLE ') 200
CBLDCMCF('TEMPGET') 222
CBLDCMCF('TEMPPUT') 223
CBLDCMCF('TIMERCAN') 244
CBLDCMCF('TIMERSET') 244
CBLDCMCF('TLSCN ') 191
CBLDCMCF('TLSCOM ') 190
CBLDCMCF('TLSLE ') 200
CBLDCMCF('TLSLN ') 198
CBLDCMCF('TOFLN ') 198
CBLDCMCF('TONLN ') 198
CBLDCPRF('PRFGETN ') 186
CBLDCPRF('PRFPUT ') 186
CBLDCRAP('CONNECT') 117
CBLDCRAP('DISCNCT') 117
CBLDCRPC('CALL ') 72
CBLDCRPC('CLTSEND') 84
CBLDCRPC('DISCARDF') 78
CBLDCRPC('DISCARDS') 79
CBLDCRPC('GETCLADR') 83
CBLDCRPC('GETERDES') 84
CBLDCRPC('GETGWADR') 48
CBLDCRPC('GETSVPRI') 83
CBLDCRPC('GETWATCH') 83
CBLDCRPC('OPEN ') 12, 17, 23
CBLDCRPC('POLLANYR') 76
CBLDCRPC('SETSVPRI') 82
CBLDCRPC('SETWATCH') 83
CBLDCRPC('SVRETRY ') 91
CBLDCRSV('MAINLOOP') 17, 32
CBLDCTAM('ERS ')('ERSR')('BRS ')('BRSR') 315
CBLDCTAM('FxxR')('FxxU')('VxxR')('VxxU') 315
CBLDCTAM('GST ') 318
CBLDCTAM('INFO') 319
CBLDCTAM('MFY ')('MFYS')('STR ')('WFY
')('WFYS')('YTR ') 315
CBLDCTRN('BEGIN ') 121
CBLDCTRN('C-COMMIT') 121
CBLDCTRN('C-ROLL ') 121
CBLDCTRN('INFO ') 153
CBLDCTRN('U-COMMIT') 121
CBLDCTRN('U-ROLL ') 121
CBLDCUTO('T-STATUS') 70
CBLDCXAT('CONNECT') 184
CCLSEVT 255, 275
CERREVT 255, 272

chained mode 122
chained RPC 135
on permanent connection 118
starting 87
terminating 87
time monitoring 88
to servers that receive requests from socket 88
using remote API facility 118
client UAP 3
acquiring node address of 83
client user program 8
client/server mode communication using OSI TP 182
closing TAM table 317
cluster/parallel mode 408
COBOL
coding 30
language 28
COBOL language template 420, 469
COBOL-UAP creation program 30
COBOL/2 30
COBOL85 30
coding 28
C or C++ 28
COBOL 30
command 156
commit optimization 136
commit processing 135
commitment 7, 122
communication data type 384
communication destination specified, RPC with 96
communication event 253
communication paradigm 370
communication through TxRPC interface 400
compilation 34, 35
conditions for
accessing TAM table 314
resending message 226
using multinode facility function 416
configuration
DAM file 286
MHP 19
SPP 14
TAM file 313
connection
coding examples for re-establishing or forcibly releasing 194
establishing 191
establishing and releasing 191
releasing 192
connection mode 116
cont 209
continuous-inquiry-response mode 207
continuous-inquiry-response processing 222
terminating 223
continuous-inquiry-response type 208, 222
conventions
abbreviations for products iv
acronyms ix
diagrams xi
fonts and symbols xiii
KB, MB, GB, and TB xiv
version numbers xv
conversational service paradigm 370, 379
COPNEVT 255, 273
creating
application program 26
physical file 300
stub 31, 33
TAM file 342
CUP 8

D

DAM file 286
configuration of 286
locking 301
logical shutdown and release of 289
referencing status of 296
DAM file service 46, 286
DAM sample 420, 440
damload command 286
data compression facility 92
data file part 356
data part 313
data transfer 74
database management system, accessing 358
dc_adm_call_command() 155
dc_adm_complete() 12, 164

dc_adm_get_nd_status() 411
dc_adm_get_nd_status_begin() 410
dc_adm_get_nd_status_done() 410
dc_adm_get_nd_status_next() 410
dc_adm_get_node_id() 414
dc_adm_get_nodeconf_begin() 414
dc_adm_get_nodeconf_done() 414
dc_adm_get_nodeconf_next() 414
dc_adm_get_sv_status() 413
dc_adm_get_sv_status_begin() 412
dc_adm_get_sv_status_done() 412
dc_adm_get_sv_status_next() 412
dc_adm_status() 165
dc_clt_accept_notification() 84
dc_clt_chained_accept_notification() 84
dc_dam_bseek() 298, 301
dc_dam_close() 288
dc_dam_create() 300
dc_dam_dget() 298, 301
dc_dam_dput() 298
dc_dam_end() 304
dc_dam_get() 297, 301
dc_dam_hold() 289
dc_dam_iclose() 297, 301
dc_dam_iopen() 297
dc_dam_open() 288
dc_dam_put() 298, 301
dc_dam_read() 289, 303
dc_dam_release() 289
dc_dam_rewrite() 289
dc_dam_start() 304
dc_dam_status() 296
dc_dam_write() 289, 303
dc_gwf_mainloop() 32
dc_ist_close() 355
dc_ist_open() 354
dc_ist_read() 354
dc_ist_write() 354
dc_jnl_ujput() 176
dc_lck_get() 361, 364
dc_lck_release_all() 362, 363
dc_lck_release_byname() 362, 363
dc_log_notify_close() 180
dc_log_notify_open() 180
dc_log_notify_receive() 180
dc_log_notify_send() 180
dc_logprint() 169
dc_mcf_adltap() 199
dc_mcf_ap_info() 51
dc_mcf_ap_info_uoc() 51
dc_mcf_close() 23
dc_mcf_commit() 228
dc_mcf_contend() 207, 223
dc_mcf_execap() 233
dc_mcf_mainloop() 23, 32
dc_mcf_open() 23
dc_mcf_receive() 206, 207, 216, 233, 276
dc_mcf_recvsync() 208, 219
dc_mcf_reply() 207, 217
dc_mcf_resend() 226
dc_mcf_rollback() 229
dc_mcf_send() 208, 217
dc_mcf_sendrecv() 208, 219
dc_mcf_sendsync() 208, 219
dc_mcf_tactcn() 191
dc_mcf_tactle() 200
dc_mcf_tdctcn() 191
dc_mcf_tdctle() 200
dc_mcf_tdlqle() 200
dc_mcf_tempget() 222
dc_mcf_tempput() 223
dc_mcf_timer_cancel() 244
dc_mcf_timer_set() 244
dc_mcf_tlscn() 191
dc_mcf_tlscom() 190
dc_mcf_tlsle() 200
dc_mcf_tlsln() 198
dc_mcf_tofln() 198
dc_mcf_tonln() 198
dc_prf_get_trace_num() 186
dc_prf_get_utrace_put() 186
dc_prf_utrace_put() 186
dc_rap_connect() 117
dc_rap_disconnect() 117
dc_rpc_call() 72
dc_rpc_call_to() 96
dc_rpc_cltsend() 84
dc_rpc_discard_further_replies() 78

dc_rpc_discard_specific_reply() 79
dc_rpc_get_callers_address() 83
dc_rpc_get_error_descriptor() 84
dc_rpc_get_gateway_address 59
dc_rpc_get_service_prio() 83
dc_rpc_get_watch_time() 83
dc_rpc_mainloop() 17, 32
dc_rpc_open() 12, 17, 23
dc_rpc_poll_any_replies() 76
dc_rpc_service_retry() 91
dc_rpc_set_service_prio() 82
dc_rpc_set_watch_time() 83
dc_rts_utrace_put() 187
dc_tam_close() 317
dc_tam_delete() 315
dc_tam_get_inf() 318
dc_tam_open() 315
dc_tam_read() 315
dc_tam_rewrite() 315
dc_tam_status() 319
dc_tam_write() 315
dc_trn_begin() 121, 232
dc_trn_chained_commit() 121, 122
dc_trn_chained_rollback() 121, 123
dc_trn_info() 153
dc_trn_unchained_commit() 121, 122
dc_trn_unchained_rollback() 121, 123
dc_uto_test_status() 70
dc_xat_connect() 184
DCADM.cbl 469
DCDAM.cbl 469
DCDMB.cbl 469
DCIST.cbl 469
DCJNL.cbl 469
DCJUP.cbl 469
DCLCK.cbl 469
DCLOG.cbl 469
DCMCF.cbl 469
DCPRF.cbl 469
DCRAP.cbl 469
DCRPC.cbl 469
DCRSV.cbl 469
dcsvstart command 12, 17, 22
dcsvstop command 12, 17, 23
DCTAM.cbl 469
DCTRN.cbl 469
DCUTO.cbl 469
DCXAT.cbl 469
deadlock
    notes on avoiding 366
    UAP response to 366
deadlock information 303, 367, 481
    output format of 481
delvcmd command 420, 467
descriptor 77
detecting user server status 165
diagram conventions xi
directory containing UAP 36
disposal in case of heuristic situation 153
DNS domain name 98
domain qualification, service request with 98
during operation
    MHP 22
    SPP 17
    SUP 11

E

entry point 101, 105
environment 36
    setting up 36
ERREVT1 253, 261
ERREVT2 229, 235, 244, 254, 262
ERREVT3 229, 244, 254, 263
ERREVT4 235, 254, 265
ERREVTA 254, 266
error event 253
event reception 383
event that reports
    detection of invalid application name 253
    discarding of message 254
    discarding of timer-start message 254
    discarding of unprocessed send message 254
    error 255
    send completion 255
    send error 254
    status 255
    UAP abnormal termination 254
EX 302, 329, 361

exclusive mode 302, 329, 361
executing operation command 155
exit routines for determining timer start
inheritance 242
extended internode load-balancing facility 42
extended RM registration definition 359

F

facilities
    TP1/Message Control 189
    TP1/Multi 407
    user data 285
facility for user timer monitoring 244
file descriptor 288, 297, 301
file-based lock 302
first retrieval 316
first segment 216
font conventions xiii
functional differences between APIs and operation commands
    application-related operations 199
    connection establishment and release 193
    MCF communication service operations 190
    shutdown and release of logical terminals 201
    start and terminate acceptance of connection establishment requests 198
functions
    available in communication modes 209
    available in operations 202
    list of 47

G

GB meaning xiv

H

hash format 313
header area 215
heuristic decision 153

I

IDL compiler 404
IDL file 404
IDL-only TxRPC 400
immediate start 234
index part 313, 356
initializing/recreating block 298
input parameter 74
input parameter length 74
input source logical terminal name 237
inputting/outputting
    arbitrary block 298
    multiple blocks collectively 289, 298
    multiple records collectively 316
inquiry-response mode 206
insufficient table pool 362
interchangeability of DAM and TAM services 312, 342
Interface Definition Language 404
intermediate segment 216
internode load-balancing facility 7, 40
internode shared table 350, 351
    accessing 354
    closing 355
    environment for access to 352
    lock for 355
    opening 354
    structure of 354
interval timer start 234
ISAM 356
ISAM file service 47, 356
ISAM/B 356
IST service 47, 350

J

jnlrput command 178
journal data editing 46, 178

K

K&R format 28
KB meaning xiv

L

last segment 216
lckrminf command 367
length of response acceptance area 74
library function 29

XATMI interface 371
library functions, list of 47
linkage 34, 35
load balancing 6
load balancing and scheduling 37
lock 329, 361
for internode shared table 355
for reference 302, 329, 361
for resource 47
for update 302, 329, 361
in online mode and offline mode 303
resource 361
TAM table 329
lock migration 363
lock mode 302, 329, 361
lock test 364
lock unit 302, 329
locking 301, 309
DAM file 301
unrecoverable DAM file 309
log service definition 169
logcat command 169
logical file 287
logical file name 287
logical message 215
logical messages and segments 215
logical terminal
deleting output queue of 200
displaying status of 200
shutting down or releasing 200

M

main function 14, 19
main program 14
mainframe 2
management
online tester 47
system operation 155
manager 402
manual startup 73
master scheduler daemon 44
maximum lock wait time, specifying 362
MB meaning xiv
MCF 5
MCF application definition 22
MCF communication process 233, 279, 281
MCF communication process identifier 242
MCF communication service operations 190
MCF event 22, 253
MCF event information 276
MCF event that reports
detection of invalid application name 261
discarding of message 262
discarding of timer-start message 265
discarding of unprocessed send message 266
error 272
establishing connection 273
releasing connection 275
send completion 271
send error 269
UAP abnormal termination 263
MCF online tester 69
MCF process 281
MCF sample 420, 462
MCF service 5
MCF transaction control 228
mcftendct command 223
mcfuevt command 242
message
resending 226
structure of 215
message communication mode 206
message exchange mode
transaction processing 7
UAP used for communication in 8
message exchange processing 205
message exchanging 46
message format 276
message handling program 8, 18
message log
outputting 169
outputting, from application program 169
message log file 169
message log notification, receiving 180
message log output 46
message processing, synchronous 218
message queue 5
message queue interface 6

message queuing mode 5
MHP 8, 18, 205
configuration 19
during operation 22
nontransaction attribute 243
rollback processing 229
starting 21
starting, using command 242
terminating 23
transaction control 228
MHP for MCF event 253
MHP processing, outline of 23
monitoring service function execution time 93
MQI 6
multi-scheduler daemon 44
multi-scheduler facility 43
examples of system configurations requiring consideration of 488
RPC 94
multinode area 410
multinode facility 47, 408
conditions for using functions of 416
multinode subarea 410
multiserver 7, 37
multiserver load balance 38

N

namdomainsetup command 99
name used when DAM file is accessed 288, 297
name used when TAM table is accessed 315
nesting service 81
NETM 172
NEXT retrieval 316
no-access optimization 146
noans 209, 256
node 3
node identifier 413
non-automatic connection mode 117
noninquiry-response mode 207
nonresident process 17, 22, 38
nonresponse type 208
nonresponse-type RPC 75, 81
nontransaction attribute 127
MHP 243
nontransactional RPC 82
using, from transaction process 82, 129
notes
access from UAP 287
adding and deleting TAM records 343
avoiding deadlock 366
transaction processing 153

O

offline tester 69
offline work, UAP that handles 8
one-phase optimization 142
online tester 69, 391
management 47
online transaction processing 2
opening TAM table 315
OpenTP1 2
samples 419
OpenTP1 client facility 8
OpenTP1 node identifier, acquiring 413
OpenTP1 node status, acquiring 410
OpenTP1 response to deadlock 366
operation command, executing 155
optimization using chained RPC 151
originator 380
OSI TP 4, 370
client/server mode communication using 182
outline
MHP processing 23
of access to DAM files 287
remote procedure call mode 74
SPP processing 17
SUP processing 12
output format
deadlock information 481
timeout information 483
undecided transaction information 476
used with TP1/FS/Table Access 486
output message, user exit routine that edits 252

P

parallel processing 76
parallel_count 38
PC 2

performance verification trace, acquiring 186
permanent connection 116
personal computer 2
physical file 286
    creating 300
physical file name 286
point of entry 101, 105
position in network 3
posting information about current transaction 153
PR 302, 329, 361
prepare optimization 139
prepare processing 135
prf trace 186
process 37
process load balancing 39
process multiserver 86
process setup method 38
process type 401
processing, continuous-inquiry-response 222
program, application 8
putenv PATH 155

Q

queue-receiving server 38

R

random access 356
RAP-processing client 111
RAP-processing listener 111
RAP-processing server 111
read-only optimization 144
real-time acquisition item definition template 420
real-time statistical information acquisition 46, 187
receive-only mode 207
receiving
    asynchronous-response-type RPC
    response 76
    message 216
    message log notification 180
    of processing result, rejecting 78
    temporary-stored data 222
record 313, 371
record input/update/addition/deletion, procedure for 315
record-based lock 329
recursive call 90
reference response waiting interval 83
referencing/updating block, procedure for 289
registering UAP excutable file 36
rejecting receiving of processing result 78
relationship between application programs and communication mode 2
relationship between asynchronous-response-type RPC and synchronization point 130
relationship between chained RPC and synchronization point 132
relationship between nonresponse-type RPC and synchronization point 132
relationship between remote procedure call and process for executing service 86
relationship between remote procedure call mode and synchronization point 129
relationship between RPC and process 86
relationship between RPC and transaction attribute 128
relationship between synchronous-response-type RPC and synchronization point 129
relationship between transaction and TAM access 319
relative block number 289
releasing resource from lock 362
remote API facility 111
    chained RPC 118
remote procedure call 3, 46, 72, 402
    transferring data through 73
remote procedure call mode, outline of 74
report data to CUP unidirectionally 84
report error event at communication event failure 262, 263
request/response service paradigm 370, 375
resending
    message 226
    message, condition for 226
resident process 17, 22, 38
resource 361
    lock 361
resource manager 358
resources which can be put under lock 361

response length 74
response storage area 74
response type 208
response waiting interval of service request, referencing and changing 83
response-type RPC 74
responses to occurrence of deadlock 366
restriction on using TX_ function 396
retrieve all record 316
retry 91
return value 29
rollback 7, 123
    in chained mode 123
    in unchained mode 123
    optimization 148
    processing 229
rollback_only status 124
rolled back 220, 229
RPC 3, 72
    communication destination specified 96
    multi-scheduler facility 94
RPC interface definition file 33
RPC modes 74
RPC trace 391
rpc_service_retry_count 91

S

sample scenario template 420
samples 420
    OpenTP1 419
schedule priority 39
    setting 82
SCMPEVT 255, 271
sector length 286
segment 215
    structure 215
send message, user exit routine that edits sequential number of 251
sending message 217
sequential access 356
SERREVT 254, 269
server 3
server UAP 3
    creating 389
servers that receive request from socket 38, 83
service 3
service function 14, 19
    relationship to stub 100
service function execution time, monitoring 93
service group name 37, 72
service name 37, 72
service program 14
service providing program 8, 13
service request
    referencing and changing response waiting interval of 83
    with domain qualification 98
service request method 371
service using program 8, 10
service_priority_control 83
setting
    environment 36
    schedule priority 82
shared mode 302, 329, 361
source file 33, 34, 35
specification
    for awaiting unlocking 329
    for message to be resent 226
    of maximum lock wait time 362
    of sample program 455
    of transaction attribute 228
    of waiting to be released from lock 303, 310
specification of sample program 455
SPP 8, 13
    configuration 14
    during operation 17
    for communication event 183
    starting 16
    terminating 17
SPP processing, outline of 17
SQL statements 13
starting
    application programs 233
    chained RPC 87
    MHP 21
    MHP for MCF event 22
    MHP using command 242
    SPP 16

SUP 11
statistical information 343, 391
when TAM service is used 343
status code 30
stbmake command 33
structure 371
internode shared table 354
message 215
segment 215
structured query language 28
stub 32, 101, 105
creating 31, 33
relationship to service function 100
type 32
using, to acquire service functions (SPP) 102
subordinator 380
subtype 384
SUP 8, 10
during operation 11
starting 11
terminating 12
SUP processing, outline of 12
symbol conventions xiii
synchronization point 7, 122
acquiring 122
synchronization point acquisition 121
synchronous exchange mode 208
synchronous message exchange processing 219
synchronous message processing 218
rollback 220
time monitoring of 219
synchronous message receive processing 219
synchronous message send processing 219
synchronous receive mode 208
synchronous send mode 208
synchronous-response-type RPC 75
time monitoring of 75
system configurations that requires consideration of multi-scheduler facility 488
system journal file 176
system operation management 46, 155

T

table descriptor 315, 354
table-based lock 329
TAM file 313
configuration of 313
creating 342
TAM file service 46, 313
TAM record, notes on adding and deleting 343
TAM sample 420, 447
TAM table 313
accessing 315
closing 317
lock 329
opening 315
TAM table access facility without table-based lock 331
TAM table information, acquiring 319
TAM table name 315
TAM table status, acquiring 318
tamcre command 342
TB meaning xiv
TCP/IP 4, 370
template 31, 469
temporary-stored data 222
receiving 222
updating 223
terminating
chained RPC 87
continuous-inquiry-response processing 223
MHP 23
SPP 17
SUP 12
test status off user server 70
tester 69
time monitoring 244
asynchronous-response-type RPC 77
chained RPC 88
of nontransaction attribute MHP 244
synchronous message processing 219
synchronous-response-type RPC 75
transaction processing 154
TX_ function 398
time point timer start 234
timeout information 367, 481
output format of 483
timer start 234

timer start inheritance, definition of 241
TP1/Client 8
TP1/FS/Direct Access 286
TP1/FS/Table Access 313
    output format used with 486
TP1/LiNK 2, 71
TP1/Message Control 5, 205
    facility 189
TP1/Message Control/Tester 69
TP1/Message Queue 6
TP1/Multi
    facility 407
TP1/Multi facility 407
TP1/NET/HNA-NIF 233
TP1/NET/Library 5, 182
TP1/NET/OSI-TP-Extended 182, 370
TP1/Offline Tester 69
TP1/Online Tester 69
TP1/Server Base 2, 71
TP1/Shared Table Access 350
tpacall() 376
TPADVERTISE 389
tpadvertise() 389
tpalloc() 388
TPCALL 376
tpcall() 375
TPCONNECT 380
tpconnect() 380
TPDISCON 381
tpdiscon() 381
tpfree() 388
TPGETRPLY 376
tpgetrply() 376
tprealloc() 388
TPRECV 380
tprecv() 380
TPRETURN 381, 389
tpreturn() 381, 389
TPSEND 380
tpsend() 380
tpservice() 389
tpstbmk command 33
TPSVCSTART 389
tptypes() 388
TPUNADVERTISE 390
tpunadvertise() 389
transaction 7
transaction attribute 82, 127, 402
    specification of 228
transaction control 46, 121, 393, 402
    MHP 228
transaction optimization 129, 135
transaction processing 7
    in message exchange mode 7
    notes on 153
    relationship with 183
    time monitoring 154
    with UAP in client/server mode 7
transaction start 121
transaction statistical information 343
transaction timeout 378, 382
transaction_mandatory 402
transaction_optional 402
transferring data though remote procedure call 73
transition of user server status
    server that receive request from socket (SPP) 168
    SPP, MHP 167
    SUP 166
tree format 313
trnlnkrm command 359
trnmkobj command 359
two-phase commit 135
TX interface 47, 393
TX_ function 394, 402
    restriction on using 396
    time monitoring with 398
tx_begin() 395
tx_close() 395
tx_commit() 395
tx_info() 396
tx_open() 395
tx_rollback() 395
tx_set_commit_return() 396
tx_set_transaction_control() 396
tx_set_transaction_timeout() 396
TXBEGIN 395
TXCLOSE 395

TXCOMMIT 395
txidl command 404
TXINFORM 396
TXOPEN 395
TXROLLBACK 395
TxRPC interface, communication through 400
TXSETCOMMITRET 396
TXSETTIMEOUT 396
TXSETTRANCTL 396
type 384
    stub 32
type of application 208
typed buffer 371, 384
typed record 371, 384

U

UAP 2
    in client/server mode 3
    in client/server mode, transaction processing with 7
    in message exchange mode 4
    registering 36
    that can be tested 70
    that can call transaction control function 122
    that handles offline work 8, 24
    that initializes user file 8
    used for communication in message exchange mode 8
UAP executable file name 37
UAP name 37
UAP process 37
UAP requiring stub 33
UAP shared library 14
    creating 36
UAP tester facility 69
UAP trace 390
UAP with transaction attribute 127
UCMDEVT 242
UJ 176
UJ record 176
unchained mode 122
undecided transaction information 476
    output format of 476
unprocessed message 276
unrecoverable DAM file 287
unrecoverable DAM file locking range 309
updating temporary-stored data 223
user data facility 285
user exit routine 248
    that determines application name 250
    that determines inheriting timer-start message 251
    that edits input message 250
    that edits output message 252
    that edits sequential number of send message 251
user file, UAP that initializes 8
user journal acquisition 46, 176
user server 3, 36
user server name 36, 37
user server process 37
user server status
    acquiring 411
    detecting 165

V

VCLSEVT 255, 275
VERREVT 255, 272
version number conventions xv
VOPNEVT 255, 273

W

WAN 5
workstation 2
WS 2

X

X/Open
    compliant API 369
    inter-application communication 399
X_C_TYPE 385
X_COMMON 385
X_OCTET 385
XA interface 358
XATMI communication service 182
XATMI interface 47, 183, 370
    library function 371

XATMI interface definition 385
XATMI interface definition file 33

Reader's Comment Form

We would appreciate your comments and suggestions on this manual. We will use these comments to improve our manuals. When you send a comment or suggestion, please include the manual name and manual number. You can send your comments by any of the following methods:

- Send email to your local Hitachi representative.
- Send email to the following address:
  WWW-mk@itg.hitachi.co.jp
- If you do not have access to email, please fill out the following information and submit this form to your Hitachi representative:

Manual name:	
Manual number:	
Your name:	
Company or organization:	
Street address:	
Comment:	

(For Hitachi use)