Symantec NetBackup™ DataStore SDK Programmer's Guide for XBSA 1.1.0

Release 7.6

Symantec

NetBackup™ DataStore SDK Programmer's Guide for XBSA 1.1.0

Documentation version: 7.6

Legal Notice

Symantec Corporation
350 Ellis Street
Mountain View, CA 94043

http://www.symantec.com

Printed in the United States of America.

10 9 8 7 6 5 4 3 2 1

Technical Support

Symantec Technical Support maintains support centers globally. Technical Support's primary role is to respond to specific queries about product features and functionality. The Technical Support group also creates content for our online Knowledge Base. The Technical Support group works collaboratively with the other functional areas within Symantec to answer your questions in a timely fashion. For example, the Technical Support group works with Product Engineering and Symantec Security Response to provide alerting services and virus definition updates.

Symantec's support offerings include the following:

- A range of support options that give you the flexibility to select the right amount of service for any size organization
- Telephone and/or Web-based support that provides rapid response and up-to-the-minute information
- Upgrade assurance that delivers software upgrades
- Global support purchased on a regional business hours or 24 hours a day, 7 days a week basis
- Premium service offerings that include Account Management Services

For information about Symantec's support offerings, you can visit our website at the following URL:

www.symantec.com/business/support/

All support services will be delivered in accordance with your support agreement and the then-current enterprise technical support policy.

Contacting Technical Support

Customers with a current support agreement may access Technical Support information at the following URL:

www.symantec.com/business/support/

Before contacting Technical Support, make sure you have satisfied the system requirements that are listed in your product documentation. Also, you should be at the computer on which the problem occurred, in case it is necessary to replicate the problem.

When you contact Technical Support, please have the following information available:

- Product release level

- Hardware information
- Available memory, disk space, and NIC information
- Operating system
- Version and patch level
- Network topology
- Router, gateway, and IP address information
- Problem description:
  - Error messages and log files
  - Troubleshooting that was performed before contacting Symantec
  - Recent software configuration changes and network changes

Licensing and registration

If your Symantec product requires registration or a license key, access our technical support Web page at the following URL:

www.symantec.com/business/support/

Customer service

Customer service information is available at the following URL:

www.symantec.com/business/support/

Customer Service is available to assist with non-technical questions, such as the following types of issues:

- Questions regarding product licensing or serialization
- Product registration updates, such as address or name changes
- General product information (features, language availability, local dealers)
- Latest information about product updates and upgrades
- Information about upgrade assurance and support contracts
- Information about the Symantec Buying Programs
- Advice about Symantec's technical support options
- Nontechnical presales questions
- Issues that are related to CD-ROMs or manuals

Support agreement resources

If you want to contact Symantec regarding an existing support agreement, please contact the support agreement administration team for your region as follows:

Asia-Pacific and Japan	customercare_apac@symantec.com
Europe, Middle-East, and Africa	semea@symantec.com
North America and Latin America	supportsolutions@symantec.com

Contents

Chapter 1

Introduction to NetBackup XBSA

This chapter includes the following topics:

About Introduction to NetBackup XBSA

Applications or facilities needing data storage management for backup or archive purposes can use the NetBackup XBSA Application Programming Interface (API) to create a backup or archive application that communicates with NetBackup.

What is NetBackup XBSA?

XBSA is an Open Group Technical Standard defining a Backup Services API (XBSA). The XBSA specification consists of source procedure calls, type definitions, data structures, and return codes to be used by client applications to use a backup service, NetBackup, to store and manage data.

The NetBackup XBSA is an API to NetBackup developed to the XBSA specifications. The NetBackup XBSA interface has extended the XBSA specifications to make it

easier to use and enhance performance when used with NetBackup. Exceptions are noted throughout the document.

See “Resources” on page 13.

NetBackup XBSA is provided as a Software Developers Kit (SDK) that includes the header files and libraries required to create an XBSA application.

What does NetBackup XBSA do?

The NetBackup XBSA interface allows an XBSA application to create, query, retrieve, and delete data objects using NetBackup for data storage. The operations on the objects use the rules and policies defined and enforced by NetBackup. Examples of these rules and policies include what type of media the objects are stored on, number of copies, retention policies, scheduled operations, etc.

Objects are CREATED and RETRIEVED as a stream of data. Each object also has a set of attributes that are used to describe the object. These attributes include a CopyId, created by the NetBackup XBSA interface, which uniquely defines the object. Other attributes are specified and used by the XBSA application to describe the object. When retrieving an object, the object is returned as a data stream and it is up to the XBSA application to restore it to its original form.

An XBSA application can also QUERY the NetBackup XBSA interface for objects that it owns. This query is based on a subset of the attributes that were specified. The result of a query is a list, possibly empty, of objects and their attributes.

Objects can also be DELETED when they are no longer needed by the XBSA application. Deleting an object prevents it from being retrieved or queried but does not necessarily delete the data. When the actual data gets deleted is a function of NetBackup.

Terminology

Fundamental terms necessary to understand this NetBackup XBSA are described below.

Table 1-1 XBSA Terms

Term	Definition
XBSA Application	Application-specific software that uses the NetBackup XBSA API to request NetBackup services. Typically an XBSA Application is tightly bound to a user application (such as a DBMS) or an operating system service (such as a file system).

Table 1-1 XBSA Terms *(continued)*

Term	Definition
NetBackup XBSA Interface	The NetBackup software that communicates with NetBackup to carry out the functions defined by this specification.
NetBackup XBSA Environment	The NetBackup XBSA Environment is the environment that exists between the NetBackup XBSA Interface and the XBSA Application. This environment is defined by a NetBackup XBSA session. NetBackup XBSA Environment variables are used to pass specific NetBackup information between the XBSA Application and the NetBackup XBSA Interface. Setting platform environment variables (such as `getenv` or `setenv`) has no effect on the NetBackup XBSA Environment.
NetBackup XBSA Session	A NetBackup XBSA session is a logical connection between a XBSA Application and NetBackup XBSA Interface. A session begins with a call to BSAInit() and ends with a call to BSATerminate(). Nested sessions are not supported.
NetBackup XBSA Object	The NetBackup XBSA API uses an object-based paradigm. Every data object visible and transferred at the NetBackup XBSA Interface is a NetBackup XBSA Object. It is up to the XBSA Application to define the objects that it will backup and restore.

Important concepts

To get the most out of using the NetBackup XBSA interface, a working knowledge of NetBackup is required. Allowing the XBSA application to control some of the NetBackup concepts such as policy, schedule, timeouts, multiplexing, etc., will allow the XBSA application to be more robust and perform better in a NetBackup environment. Other items, such as storage units, determine where data gets stored, which could affect the XBSA application.

Note, however, that the NetBackup XBSA interface does not provide an interface for managing the configuration, media, jobs, etc. These types of operations must be done through other NetBackup command line or graphical interfaces.

Resources

The NetBackup XBSA API specification is based on the Open Group Technical Standard for Systems Management: Backup Services API (XBSA) Document Number: C425. More information on this standard can be found at the following URL:

http://www.opengroup.org/products/publications/catalog/c425.htm.f

Chapter 2

How to set up the SDK

This chapter includes the following topics:

System requirements

The following items are needed before setting up the SDK.

- Supported systems. Refer to the Database Agent Compatibility list for NetBackup Enterprise Server for a list of platforms that are supported with XBSA. This list is available from the following Web site: http://entsupport.symantec.com
- ANSI-compatible compiler.
- To develop an application, you need NetBackup, a DataStore License Key, and the NetBackup DataStore SDK installed.
- To run an application, you need a NetBackup client installed (on client running XBSA application) and the DataStore License Key (on NetBackup server)

Installing the SDK

The NetBackup for DataStore SDK is released on a separate CD from the rest of NetBackup. You must have this CD to install the SDK. Once installed, the files should be moved to the environment where the development of the XBSA application is to be done.

Installation requirements

The following items are required before installing the SDK:

- NetBackup server software is installed and operational on the server where the SDK is to be installed.
- Adequate disk space (approximately 20 M) on the server must be present to receive the software.

Installation instructions for UNIX platforms

To install the SDK on UNIX platforms

1. Log on as the root user on the computer.

   If you are already logged on, but are not the root user, execute the following command.

   ```
   su - root
   ```

2. Verify that a registered and valid license key for resides on the master server.

   To view or add license keys, perform one of the following:

   - Run the following command:

     ```
     /usr/openv/netbackup/bin/admincmd/get_license_key
     ```

     If you run the `get_license_key` command from the master server, it returns the correct information by default.
     If you run the `get_license_key` command from the media server, specify the master server's host name as the machine you are querying. If you do no specify the master server's host name, the command returns licensing information about the media server.

   - Open the NetBackup Administration Console and choose **Help > License Keys**.

3. Insert the NetBackup DataStore SDK CD-ROM into the drive.

4 Change the working directory to the CD-ROM directory.

```
cd /CD_mount_point
```

5 Load and install the software by executing the install script.

```
./install
```

A prompt appears asking if the package is correct.

Answer `y`.

The SDK files are extracted into the directory *install_path*`/openv/netbackup/sdk`.

The file `version_dstore` is extracted into the directory *install_path*`/openv/share`.

Installation instructions for Windows platforms

To install the SDK on Windows platforms

1 Insert the CD-ROM into the drive.

- On systems with AutoPlay enabled for CD-ROM drives, the install program starts automatically.
- On systems that have AutoPlay disabled, click the **Start** button and choose **Run**. Type `D:\Autorun\AutoRunI.exe`, where D:\ is your CD-ROM drive.

2 Follow the prompts throughout the wizard.

Uninstalling the SDK

The NetBackup for DataStore SDK is delivered in native packaging format. Remove the NetBackup for DataStore SDK by executing the native command appropriate for your operating system:

- AIX: `installp -u SYMCnbsds`
- HP-UX: `swremove SYMCnbsds`
- Linux: `rpm -e SYMCnbsds`
- Solaris: `pkgrm SYMCnbsds`

Windows: On the **Windows Control Panel**, select **Add/Remove Programs**. Select **Symantec NetBackup - DataStore**, then click **Remove**.

Configuration

Creating an XBSA application using the NetBackup XBSA SDK should require a minimum of setup. The SDK is installed as read only in the NetBackup directory. It is recommended that the files that are going to be used be moved to the development environment of the application.

The sample directory provides a Makefile for UNIX platforms and one for Windows platforms. They will create valid executables for the sample programs, but they should be used as guides only and the developers should use the compile options and libraries that are optimal for their application. The XBSA libraries and header files themselves do not require any special options.

Description of XBSA SDK package

The NetBackup SDK contains the libraries with the XBSA interfaces for each of the platforms that the SDK supports. There are header files that are required to compile an XBSA Application. The SDK is installed in the NetBackup directory, either `/usr/openv/netbackup/sdk/DataStore/XBSA` on UNIX or *`install_directory`*`\VERITAS\NetBackup\sdk\DataStore\XBSA` on Windows. This directory will contain all files necessary to build an XBSA Application.

The package contains the following directories.

Table 2-1 SDK/DataStore/XBSA Directories

Directory	Description
samples	Contains sample programs and scripts.
lib	Contains the library files for each supported system.
include	Contains the header files.

Library files

The NetBackup XBSA SDK contains the archive libraries for each of the systems. Installed with the NetBackup client is an XBSA shared object library. This allows the developer to choose the method of binding for each application. Both of these libraries contain all XBSA functions and all external references.

The XBSA libraries are found in the directory `/usr/openv/netbackup/sdk/DataStore/XBSA/lib`. In this directory is a directory for each hardware type. Within each of these directories is a directory for each supported operating system level. For UNIX operating systems, there is the

libxbsa.a library. For the Windows operating systems, there is both an xbsa.lib and a xbsas.lib. The xbsa.lib was generated when creating the xbsa.dll and xbsas.lib is a full static library.

Header files

There are two header files that are released with the SDK. These should be used when compiling the XBSA Application. These header files are found in directory /usr/openv/netbackup/sdk/DataStore/XBSA/include.

Table 2-2 Header Files

File	Description
xbsa.h	Header file that contains the XBSA defined structures.
nbbsa.h	Header file that contains NetBackup specific definitions for the NetBackup XBSA Interface.

Chapter 3

Using the NetBackup XBSA interface

This chapter includes the following topics:

Getting help with the API

While working with the API, you can obtain reference information about XBSA functions.

Sample applications are included with XBSA.

See "What the sample files do" on page 87.

NetBackup XBSA data structures

This section describes the XBSA data structures and explains how the NetBackup XBSA interface and the XBSA Application use them for creating and manipulating XBSA objects.

Object data

NetBackup XBSA Object data contains the actual data entity that is archived or backed up by an XBSA Application. The NetBackup XBSA API supports only one type of object data, namely, a variable-length, unstructured and uninterpreted byte-stream.

To a particular XBSA Application, however, the XBSA Object Data can contain an internal structure that reflects the data of the Application Object or Objects that the XBSA Application archived or backed up. In this context the XBSA Object Data can contain, for example, one of the following: a UNIX file system, a UNIX directory, a file, a document, a disk image, a data stream, or a memory dump.

Through the NetBackup XBSA Interface, object data can be stored, retrieved, or deleted, but not searched or modified. Since object data may be stored on slow (or off-line) media, it is generally not advisable for an XBSA Application to store metadata in object data, especially information that could influence a data-retrieval decision.

However, the metadata of an XBSA Object, which is stored in the catalog, may be replicated in its object data if it could enhance the performance of the restore of the object. This is an XBSA Application implementation decision.

Object descriptors

A NetBackup XBSA Object has a BSA_ObjectDescriptor, containing cataloging information and optional application-specific object metadata. Cataloging information is capable of interpretation and searching by the NetBackup XBSA Interface. Application-specific object metadata is not interpretable by the NetBackup XBSA Interface but may be retrieved and interpreted by an application. Using an object's objectName or its assigned copyId identifier, the corresponding BSA_ObjectDescriptor and object data can be retrieved through the NetBackup XBSA Interface.

A BSA_ObjectDescriptor consists of a collection of object attributes. The basic data types used for XBSA Object attributes are:

- Fixed-length character strings
- Hierarchical character strings (with a specified delimiter, and a length limit on the overall string)
- Enumerations
- Integers (with a specified range limit)
- Date-time (in a standard C TM structure) format and precision, for example, yyyymmddhhmm)

The attributes are shown in the following table:

Table 3-1 BSA_ObjectDescriptor Attributes

Attribute	Data Type	Searchable
objectOwner	(consisting of two parts)	Yes
bsa_ObjectOwner	[fixed-length character string]	
app_ObjectOwner	[hierarchical character string]	
objectName	(consisting of two parts)	Yes
objectSpaceName	[fixed-length character string]	
pathName	[hierarchical character string]	
createTime	[date-time]	Yes
copyType	[enumeration]	Yes
copyId	64-bit unsigned integer	No
restoreOrder	64-bit unsigned integer	No
resourceType	[fixed-length character string]	No
objectType	[enumeration]	Yes
objectStatus	[enumeration]	Yes
objectDescription	[fixed-length character string]	No
estimatedSize	[64-bit unsigned integer]	No
objectInfo	[fixed-length byte string]	No

Each NetBackup XBSA Object is a copy of certain application object(s):

- To preserve the semantics of the use of each copy within the BSA_ObjectDescriptor, each NetBackup XBSA Object has a copyType of either backup or archive, which is recognized by the NetBackup XBSA Interface so that the two types of objects can be managed differently and accessed separately. Note that it is up to the XBSA Application to manage these types differently, as the NetBackup XBSA Interface only keeps track of which type the object is.
- Each NetBackup XBSA Object also has an objectStatus of either most_recent or not_most_recent.

- To capture an application object's type information, the corresponding NetBackup XBSA Object may have a resourceType (for example, "filesystem") and a possibly resource-specific BSA_ObjectType (for example, BSA_ObjectType_FILE).

A XBSA Application may search for a particular NetBackup XBSA Object within a certain search scope (for example, among objects in a certain objectSpaceName) by qualifying the search on the value of the appropriate searchable attributes.

On the other hand, non-searchable, application-specific attributes may be provided by a XBSA Application for storage in the BSA_ObjectDescriptor, but the NetBackup XBSA Interface does not interpret these attributes. They are stored in the NetBackup XBSA Object attributes objectInfo, resourceType, and objectDescription.

The objectInfo field defaults to a character string. It can also be used to store binary data by using the NBBSA_OBJINFO_LEN XBSA environment variable.

Through these descriptor attributes, application-specific metadata may be stored in the catalog so that this metadata can be efficiently retrieved without retrieving the actual object data stored in the repository. These attributes can be used by a XBSA Application to maintain inter-object relationships and dependencies. Be aware though that some consideration should be given to how much data is being stored in the NetBackup Catalog. The amount of metadata stored with a few large objects can be larger than that stored for a million small objects.

Query descriptors

A BSA_QueryDescriptor is the structure that is used in the query process to find an individual or set of objects. It contains those fields from the object descriptor that are searchable. When doing a query, it is required that the enumeration fields are specified. If they are unknown, they all allow an "ANY" enumeration. It is also required to specify the objectName.pathName. Wild cards are allowed for this field and "/*" is a valid pathname for querying. The other strings in the descriptor can be empty strings, but they will still be used for comparison to find an object descriptor that matches the query descriptor. If these fields are unknown, wild cards are allowed here also. The start (createTime_from) and end (createTime_to) dates are not required.

The attributes of the BSA_QueryDescriptor are shown in the following table:

Table 3-2 BSA_QueryDescriptor Attributes

Attribute	Data Type
objectOwner	(consisting of two parts)
bsa_objectOwner	[fixed-length character string]

Table 3-2 BSA_QueryDescriptor Attributes *(continued)*

Attribute	Data Type
app_objectOwner	[hierarchical character string]
objectName	(consisting of two parts)
objectSpaceName	[fixed-length character string]
pathName	[hierarchical character string]
createTime_from	[date-time]
createTime_to	[date-time]
CopyType	[enumeration]
objectType	[enumeration]
objectStatus	[enumeration]

Note: The createTime_from and createTime_to fields are not part of the XBSA specification for the BSA_QueryDescriptor structure. The NetBackup XBSA Interface is using 2 reserved fields from the BSA_QueryDescriptor structure to allow this information to be used (if available) for the query. These fields are not required, although if the XBSA Application can specify these dates, it can, in some instances, greatly speed up query time.

Buffers

All buffers that are used by NetBackup XBSA Interface are allocated by the XBSA Application. The NetBackup XBSA Interface fills data into the buffers, but never allocates any memory that is passed back to the XBSA Application. This simplifies buffer allocation and deletion since the XBSA Application is solely responsible.

However, to allow the NetBackup XBSA Interface to influence how buffers should be allocated and to provide an interface with the ability to reserve private sections in certain buffers, the API uses several conventions.

Buffer size

For API calls that specify the size of the buffer as a separate parameter, the interface uses the following convention to signal that a buffer is not large enough and provide the XBSA Application with the means to discover what the correct size should be.

The parameter that specifies the size is a pointer, so that the NetBackup XBSA Interface can alter the parameter. The size is always in bytes. If the size is adequate and a valid buffer is given, the NetBackup XBSA Interface will copy the requested data into the buffer and set the actual size in the size parameter.

If the size is inadequate, the NetBackup XBSA Interface will not copy the data into the buffer. It will set the size parameter to the actual size of the data to be copied and return from the function call with BSA_RC_BUFFER_TOO_SMALL. This allows the XBSA Application to allocate a buffer of adequate size and to call the function again.

The functions that use this convention are BSAGetEnvironment(), NBBSAGetEnv() and BSAQueryServiceProvider().

Private buffer space

For function calls that use the BSA_DataBlock32 structure, a convention has been adopted that allows the NetBackup XBSA Interface to reserve certain portions of the buffer for its own use. There are two areas that can be reserved by the NetBackup XBSA Interface:

Header	A contiguous area starting at offset 0 (that is, the start of the buffer)
Trailer	A contiguous area that ends at the end of the buffer (that is, the tail of the buffer)

The area reserved for the XBSA Application is the:

Data Segment	A contiguous area that lies in between the Header and Trailer

To make this preference known to the XBSA Application, the NetBackup XBSA Interface may set certain parameters in the BSA_DataBlock32 structure when a data transfer is initiated. Specifically, when the XBSA Application issues either the BSACreateObject() call or the BSAGetObject() call, the BSA_DataBlock32 structure is not used for passing data but for passing the NetBackup XBSA Interface's preference. The parameters that are set by the NetBackup XBSA Interface, and their meaning, are specified in the following table:

Table 3-3 Parameters in the BSA.DATABlock32 Structure

Parameter	Preference
bufferLen == 0	The interface has no restrictions on the buffer length. No trailer portion is required.

Table 3-3 Parameters in the BSA.DATABlock32 Structure *(continued)*

Parameter	Preference
bufferLen != 0	The interface accepts buffers that are at least bufferLen bytes in length (minimum length). It also accepts larger buffers. For a BSASendData() call, the interface accepts a trailer that is as least as large as: trailerBytes >= (bufferLen - numBytes - headerBytes) For a BSAGetData() call, the interface returns a trailer that is not larger than: trailerBytes <= (bufferLen - numBytes - headerBytes)
numBytes == 0	The interface has no restrictions on the length of the data portion of the buffer.
numBytes != 0	The interface accepts (for a BSASendData() call), or returns (for a BSAGetData() call), a data segment that does not exceed numBytes bytes.
headerBytes == 0	The interface only accepts or returns buffers with no header.
headerBytes != 0	The length of the header portion of buffers accepted or returned by the interface is headerBytes bytes.
bufferPtr	Not used

Subsequent calls to BSAGetData() or BSASendData() must adhere to the preferences that were specified by the NetBackup XBSA Interface.

The NetBackup XBSA Interface can write anything into the header and trailer area of the actual buffer, as specified by the bufferPtr parameter in the BSA_DataBlock32 structure.

The NetBackup XBSA Interface has a buffer size limit of 1 Gigabyte.

Note: For NetBackup XBSA Version 1.1.0, there are no header or trailer requirements. The format documented here is defined by the XBSA specifications and may be used in the future by NetBackup.

Use of BSA_DataBlock32 in BSASendData()

For BSASendData(), all parameters in the BSA_DataBlock32 structure must be set by the XBSA Application and adhere to the NetBackup XBSA Interface preferences or the function will fail with a BSA_RC_INVALID_DATABLOCK error. The NetBackup XBSA Interface is not allowed to change any of the parameters.

The buffers being passed by BSASendData() must be full. This means that numBytes must be equal to bufferLen. The buffer for the last BSASendData() call for an object does not need to be full.

Use of BSA_DataBlock32 in BSAGetData()

For BSAGetData(), all parameters in the BSA_DataBlock32 structure must be set by the XBSA Application and adhere to the NetBackup XBSA Interface preferences or the function will fail with a BSA_RC_INVALID_DATABLOCK error. The NetBackup XBSA Interface will change the numBytes parameter setting the actual number of bytes copied into the data segment. NetBackup is not allowed to change any of the other parameters.

Shared memory

Note: Passing of data in shared memory blocks between the XBSA Application and the NetBackup XBSA Interface is not supported for NetBackup XBSA Version 1.1.0.

The BSA_DataBlock32 structure contains fields to allow the use of shared memory blocks for passing data between a XBSA Application and the NetBackup XBSA Interface. The shareId and shareOffset fields of the BSA_DataBlock32 structure are used to define shared memory buffers. NetBackup XBSA Interface version 1.1.0 does not use these fields.

NetBackup XBSA environment

The NetBackup XBSA environment is created when an XBSA Application calls BSAInit() to initiate a session. This environment only exists between the NetBackup XBSA Interface and the XBSA Application. XBSA environment variables are used to pass specific NetBackup information in both directions between the XBSA Application and the NetBackup XBSA Interface. The environment variables are generally set or modified by the XBSA Application, but the NetBackup XBSA Interface does create and/or modify some variables in order to pass information back to the XBSA Application. Setting platform environment variables (getenv or setenv) has no effect on the NetBackup XBSA environment.

There are restrictions on when some of the variables can be set/modified. Most of them can be set on the call to BSAInit(), which initiates a session. Some can also be modified within a session but outside of a transaction. And a few can be modified within a transaction. These limitations are outlined below in the descriptions for each of the variables.

Each XBSA environment variable is defined as a keyword followed by an "=" and followed by a null-terminated value. No spaces are allowed around the "=". "BSA_API_VERSION=1.1.0" is valid while "BSA_API_VERSION = 1.1.0" is not.

The functions used to create, modify, and view these environment variables are:

- BSAInit()
- BSAGetEnvironment()
- NBBSAUpdateEnv()
- NBBSASetEnv()
- NBBSAGetEnv()

These functions are defined later in the API Function Definitions section of this document.

Environment variable definitions

The following XBSA environment variables are defined as part of the XBSA specification and are accepted by the NetBackup XBSA Interface.

Table 3-4 XBSA Environment Variables

Variable Name	Description	Format
BSA_API_VERSION	Mandatory. Specifies the version of the specification that the calling XBSA Application requires. BSAQueryApiVersion() can retrieve the value of the current NetBackup XBSA Interface.	A string containing 3 numeric elements, (version, issue, level) separated by periods.
BSA_DELIMITER	Optional. The delimiter used in hierarchical character strings (default "/").	A single ASCII character.
BSA_SERVICE_PROVIDER	Optional. Identifies the XBSA implementation. BSAQueryServiceProvider() can retrieve this value.	A hierarchical character string with 3 fields.
BSA_SERVICE_HOST	Optional. Identifies a specific host system for the NetBackup Server.	A string containing a host name.

In addition to the environment variables defined in the XBSA specification, the following NetBackup XBSA environment variables are defined as part of this specification. These are specific to NetBackup and have no relevance to other XBSA implementations. See the *NetBackup System Administrator's Guide, Volume*

I, for a more complete definition of NetBackup policy, schedule, and logging. The NetBackup environment variables all are prefaced with "NB."

Table 3-5 NetBackup Environment Variables

Variable Name	Description	Format
NBBSA_CLIENT_HOST	Optional. Identifies a specific host system for the NetBackup client.	A string containing a host name.
NBBSA_DB_TYPE	Optional. This specifies a specific policy type.	A string containing the policytype.
NBBSA_FEATURE_ID	Optional. This specifies a specific NetBackup licensed feature within the DataStore policy type.	An integer value.
NBBSA_KEYWORD	Optional. If this is specified, this value will be used for the NetBackup Keyword field for this image.	A string containing a keyword value <= 100 characters.
NBBSA_LOG_DIRECTORY	Optional. Identifies the name of directory that will contain the log files of the XBSA Application.	A string containing a single directory name.
NBBSA_OBJECT_GROUP	Optional. This variable is used to define the object group owner of an object being created.	A string containing the group.
NBBSA_OBJECT_OWNER	Optional. This variable is used to define the object owner of an object being created.	A string containing the owner.
NBBSA_OBJINFO_LEN	Optional. If this variable is set before an XBSA Object is created the objectInfo field will be considered to be of this length and the object will be considered binary.	An integer value <= 256.
NBBSA_POLICY	Optional. Identifies a specific NetBackup policy to be used.	A string containing a NetBackup policy name.
NBBSA_SCHEDULE	Optional. Identifies a specific NetBackup XBSA Schedule to be used.	A string containing a NetBackup schedule name.
NBBSA_USE_OBJECT_GROUP	Optional. This variable can be set to cause the group of an object to be something other than the login user creating the object.	An integer value between 0 and 4.

Table 3-5 NetBackup Environment Variables *(continued)*

Variable Name	Description	Format
NBBSA_USE_OBJECT_OWNER	Optional. This variable can be set to cause the owner of an object to be something other than the login user creating the object.	An integer value between 0 and 4.

The following XBSA environment variables are set by the NetBackup XBSA Interface from values in the NetBackup configuration files. These environment variables are used to pass required information from NetBackup to the XBSA Application. Descriptions of these NetBackup configuration values can be found in the *NetBackup System Administrator's Guide for UNIX, Volume I*, or *NetBackup System Administrator's Guide for Windows, Volume I*.

Table 3-6 XBSA Environment Variables for NetBackup Configuration Values

Variable Name	Description	Format
NBBSA_VERBOSE_LEVEL	The verbose level of the database logs.	An integer value between 0 and 9.
NBBSA_MULTIPLEXING	The NetBackup MULTIPLEXING value.	An integer value.
NBBSA_SERVER_BUFFSIZE	The NetBackup Server Buffer Size value.	An integer value in bytes.
NBBSA_MEDIA_MOUNT_TIMEOUT	The NetBackup MEDIA_MOUNT_TIMEOUT value.	An integer value in seconds.
NBBSA_CLIENT_READ_TIMEOUT	The NetBackup CLIENT_READ_TIMEOUT value. This value can be modified by the XBSA Application.	An integer value in seconds.

Extended environment variable definitions

Table 3-7 Extended Environment Variables

Variable Name	Extended Description
BSA_API_VERSION	BSA_API_VERSION specifies the version of the XBSA specification. It is set by the XBSA Application as the version that the XBSA Application requires. This value is required to be in the environmental variable list in the call to BSAInit(), where it will be verified as a supported version of the NetBackup XBSA Interface. The current value of BSA_API_VERSION that is supported by the NetBackup XBSA Interface can be retrieved with a call to BSAQueryApiVersion(). Once BSA_API_VERSION has been set in the XBSA environment, it cannot be changed via calls to NBBSAUpdateEnv() or NBBSASetEnv(). The version supported for this feature pack is "1.1.0."
BSA_DELIMITER	BSA_DELIMITER is the delimiter used in hierarchical character strings. The NetBackup XBSA Interface sets this XBSA environment variable. The delimiter used by this feature pack is "/". This value can be retrieved by BSAQueryServiceProvider().
BSA_SERVICE_HOST	BSA_SERVICE_HOST identifies the host system for the NetBackup Server. If this variable is not provided, the currently configured server for the NetBackup Client will be used. See the *NetBackup System Administrator's Guide for UNIX, Volume I*, or *NetBackup System Administrator's Guide for Windows, Volume I*, for information on how to use the configuration file, `bp.conf`, to specify the NetBackup servers. This XBSA environment variable may be set by the XBSA Application via BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but may not be set or modified after a transaction has begun.
BSA_SERVICE_PROVIDER	BSA_SERVICE_PROVIDER identifies the XBSA implementation. The NetBackup XBSA Interface sets this XBSA environment variable. It is defined as: Symantec/NetBackup/1.1.0. BSAQueryServiceProvider() may also retrieve this value.

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
NBBSA_CLIENT_HOST	NBBSA_CLIENT_HOST identifies a specific host system as the NetBackup client. If this variable is not provided, the host the XBSA Application is running on is the client. This variable is useful for queries and restores when restoring data that was backed up from a different host than the host where the data is being restored. For backups, if the NBBSA_CLIENT_HOST is logically different from the client host the backup is being initiated from, this will result in an error, as you cannot create objects from another host. This XBSA environment variable may be set by the XBSA Application via BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but may not be set or modified after a transaction has begun.
NBBSA_CLIENT_READ_TIMEOUT	NBBSA_CLIENT_READ_TIMEOUT is used to determine or reset the NetBackup CLIENT_READ_TIMEOUT value. The NetBackup XBSA Interface creates this XBSA environment variable in the function BSACreateObject() or BSAGetObject(). After BSACreateObject(), the NBBSA_CLIENT_READ_TIMEOUT value may be reset by the XBSA Application via NBBSAUpdateEnv() or NBBSASetEnv(). Setting it at any other time will have no effect. See the *NetBackup System Administrator's Guide for UNIX, Volume I*, or *NetBackup System Administrator's Guide for Windows, Volume I*, for more information about CLIENT_READ_TIMEOUT.
NBBSA_DB_TYPE	NBBSA_DB_TYPE is an internal string representation of a NetBackup policy type. This is generally only used for NetBackup internal agents, but in certain instances may be set up for external use. If this variable is not specified, it defaults to the SDK default of DataStore policy type. If this variable is used, the NBBSA_FEATURE_ID must also be specified.
NBBSA_FEATURE_ID	NBBSA_FEATURE_ID identifies a specific NetBackup licensed feature to be used for the session. If this variable is not provided, the default DataStore feature id will be used. In general this environment variable does not need to be set, but it allows an application, working with NetBackup product management, to use a specific NetBackup license. This value may be set by the XBSA Application via BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but may not be set or modified after a transaction has begun.

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
NBBSA_KEYWORD	NBBSA_KEYWORD will allow the XBSA Application to specify a NetBackup keyword. This keyword is typically used to group images together and can speed up a search. If this variable is specified for a backup transaction, the keyword will be stored with the image. If it is specified before a query or restore transaction, the keyword will be used to help in the search process. This value may be set by the XBSA Application via BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but may not be set or modified after a transaction has begun.
NBBSA_LOG_DIRECTORY	NBBSA_LOG_DIRECTORY identifies the name of directory that will contain the log files of the NetBackup XBSA Interface and possibly for the XBSA Application. This directory will be located in `/usr/openv/netbackup/logs` on UNIX and *`install_directory`*`\VERITAS\NetBackup\Logs` on Windows. If not specified, the directory name will be `exten_client`. All debug messages from the NetBackup XBSA Interface and from function NBBSALogMsg() go to a dated log file in this directory. This value may be set by the XBSA Application via BSAInit(). It may not be modified after the call to BSAInit().
NBBSA_MEDIA_MOUNT_TIMEOUT	NBBSA_MEDIA_MOUNT_TIMEOUT is used to determine the NetBackup MEDIA_MOUNT_TIMEOUT value. The NetBackup XBSA Interface creates this XBSA environment variable in the function BSACreateObject() or BSAGetObject(). NBBSA_MEDIA_MOUNT_TIMEOUT may not be modified by the XBSA Application. See the *NetBackup System Administrator's Guide for UNIX, Volume I*, or *NetBackup System Administrator's Guide for Windows, Volume I*, for more information about MEDIA_MOUNT_TIMEOUT.
NBBSA_MULTIPLEXING	NBBSA_MULTIPLEXING the number of streams that NetBackup has been configured to accept at one time. The NetBackup XBSA Interface creates this XBSA environment variable in the function BSACreateObject() or BSAGetObject(). NBBSA_MULTIPLEXING may not be modified by the XBSA Application. See the *NetBackup System Administrator's Guide, Volume I*, for more information about multiplexing.

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
NBBSA_OBJECT_GROUP	NBBSA_OBJECT_GROUP can be used in conjunction with variable NBBSA_USE_OBJECT_GROUP to define the group ownership of an object. When NBBSA_USE_OBJECT_GROUP = VxENV_OWNER, the name defined in this string becomes the group owner of an object that is created. This group should be a valid groupname on the client. This value may be set by the XBSA Application via BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv(). It can be modified within a transaction and each object created within one transaction could have a different group.
NBBSA_OBJECT_OWNER	NBBSA_OBJECT_OWNER can be used in conjunction with variable NBBSA_USE_OBJECT_OWNER to define the ownership of an object. When NBBSA_USE_OBJECT_OWNER = VxENV_OWNER, the name defined in this string becomes the owner of an object that is created. This owner should be a valid username on the client. This value may be set by the VxBSA Application via BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv(). It can be modified within a transaction and each object created within one transaction could have a different owner.
NBBSA_OBJINFO_LEN	NBBSA_OBJINFO_LEN is used by BSACreateObject() to allow the objectInfo field of the object descriptor to contain non-ASCII values. If this variable is not specified, the objectInfo field will be treated as a NULL terminated character string. It is not required to specify this variable for a query or restore transaction. This value may be modified by the XBSA Application at any time during a backup transaction using BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv(). If the length of the objectInfo field is different for each object, it can be changed before each BSACreateObject() call.
NBBSA_POLICY	NBBSA_POLICY identifies a specific NetBackup policy to be used for the transaction. If this variable is not provided, the NetBackup configuration will be used to find the default policy to use. For backups, if a policy is configured in NetBackup on the client, that policy is used for the backup. For queries, restores, and deletes, the configured policy is not used. See the *NetBackup System Administrator's Guide for UNIX, Volume I*, or *NetBackup System Administrator's Guide for Windows, Volume I*, for information on how to create and configure a NetBackup policy. This value may be set by the XBSA Application via BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but may not be set or modified after a transaction has begun.

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
NBBSA_SCHEDULE	NBBSA_SCHEDULE identifies a specific NetBackup schedule to be used. If this variable is not provided, the NetBackup configuration will be used to find the default schedule to use. For backups, if a schedule is configured in NetBackup on the client, that schedule is used for the backup. For queries, restores, and deletes, the configured schedule is not used. See the *NetBackup System Administrator's Guide for UNIX, Volume I*, or *NetBackup System Administrator's Guide for Windows, Volume I*, for information on how to create and configure a NetBackup Schedule. This value may be set by the XBSA Application via BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but may not be set or modified after a transaction has begun.
NBBSA_SERVER_BUFFSIZE	NBBSA_SERVER_BUFFSIZE the NetBackup configured size of the NET_BUFFER_SZ. This can be used by XBSA application to help improve performance. The NetBackup XBSA Interface creates this XBSA environment variable in the function BSACreateObject() or BSAGetObject(). NBBSA_SERVER_BUFFSIZE may not be modified by the XBSA Application. See theSee the *NetBackup System Administrator's Guide for UNIX, Volume I*, or *NetBackup System Administrator's Guide for Windows, Volume I*, for more information about setting the buffer size.

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
NBBSA_USE_OBJECT_GROUP	NBBSA_USE_OBJECT_GROUP allows the agent to define the group owner of objects created with VxBSACreateObject(). The default group of an object is the login user of the process creating the object (not the primary group of the login user, but the actual login user). This variable allows the agent to specify the ownership as follows. VxLOGIN_USER 0 - Default, group field is set to the login user VxLOGIN_GROUP 1 - Group field is set to the primary group of the login user VxBSA_OWNER 2 - Group field is set to objectDescriptor->objectOwner.bsa_ObjectOwner VxAPP_OWNER 3 - Group field is set to objectDescriptor->objectOwner.app_ObjectOwner VxENV_OWNER 4 - Group field is set to value of NBBSA_GROUP_OWNER variable This value may be set by the BSA Application via BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but may not be set or modified after a transaction has begun.
NBBSA_USE_OBJECT_OWNER	NBBSA_USE_OBJECT_OWNER allows the agent to define the owner of objects created with BSACreateObject(). The default ownership of an object is the login user of the process creating the object. This variable allows the agent to specify the ownership as: VxLOGIN_USER 0 - Default, owner field is set to the login user VxBSA_OWNER 2 - Owner field is set to objectDescriptor->objectOwner.bsa_ObjectOwner VxAPP_OWNER 3 - Owner field is set to objectDescriptor->objectOwner.app_ObjectOwner VxENV_OWNER 4 - Owner field is set to value of NBBSA_OBJECT_OWNER variable This value may be set by the XBSA Application via BSAInit(), NBBSASetEnv(), or NBBSAUpdateEnv() but may not be set or modified after a transaction has begun.

Table 3-7 Extended Environment Variables *(continued)*

Variable Name	Extended Description
NBBSA_VERBOSE_LEVEL	NBBSA_VERBOSE_LEVEL is the verbose level of the NetBackup debug logs. The verbose level can be configured through both the Backup, Archive, and Restore interface or the NetBackup Administration Console. This value may be useful if the XBSA Application, using NBBSALogMsg(), wants to log different levels of messages to the NetBackup XBSA logs based on the verbose level that is configured in NetBackup. The NetBackup XBSA Interface will originally set this value in BSAInit(). The XBSA Application may reset this environment variable, using NBBSASetEnv() or NBBSAUpdateEnv(), if it wants to change the level of logging.

XBSA sessions and transactions

All operations for NetBackup must be in an XBSA session. Each session can contain one or more transactions. This section defines how XBSA sessions are defined and what can be in each transaction.

Sessions

In order to use most of the NetBackup XBSA API calls, it is necessary for a XBSA Application to set up a session with the NetBackup XBSA Interface by invoking the BSAInit() call. The functions BSAQueryApiVersion() and BSAQueryServiceProvider() may be invoked prior to calling BSAInit(). These functions are used to determine the current version of the API used by the NetBackup XBSA Interface and a string describing the provider of the NetBackup XBSA Interface, respectively, and are not dependent on being within a session.

Initialization and termination

A session is initiated by a BSAInit() call. This call sets up a session with the NetBackup XBSA Interface and creates a context, defined by handle, for the caller to be used in subsequent calls. The XBSA environment is set up within that context and remains in place until the session is terminated. Nested sessions are not permitted.

A session is terminated by a BSATerminate() call, which will release any resources acquired during the NetBackup XBSA session. If BSATerminate() is called within a transaction, the transaction is aborted.

Authentication

In NetBackup XBSA Version 1.1.0, all authentication and security is handled by NetBackup based on the login user. Object ownership is determined by the login user of the session that created the object. In order to query or restore an object, the login user doing the request must be the same user who created the object or a root administrator.

Note: NetBackup XBSA Version 1.1.0 does not validate the objectOwner and SecurityToken parameters of BSAInit(). The objectOwner fields, bsa_objectOwner and app_objectOwner, can be specified and will be stored with an object, but the login user who created the object determines the official ownership of an object. This user, or a root admin, are the only users who can query or restore this object.

Transactions

Within each session, a XBSA Application can make a sequence of calls (for example, to backup some objects, to query the set of objects it has backed up, or to restore objects). These calls must be grouped into a transaction by invoking BSABeginTxn() at the beginning of the group of calls and invoking BSAEndTxn() at the end. The latter either commits the transaction or aborts it.

If a transaction is aborted either by a BSAEndTxn() or BSATerminate() call, then the effect of all the calls made within the transaction is nullified. If a transaction is committed, then the effect of all the calls within the transaction is made permanent.

Within a single session, transactions cannot be nested and cannot overlap. Transactions are categorized into the following types:

- NetBackup XBSA Object modification transactions - in which NetBackup XBSA Objects may be created or deleted.
- NetBackup XBSA Object retrieval transactions - in which NetBackup XBSA Objects may only be queried and/or retrieved. This type of transaction provides no functional benefit for the calling XBSA Application, and is only included for completeness.

The type of a transaction is established by the first create/delete/retrieve operation performed. Attempts to mix operations in a transaction will result in a BSA_RC_INVALID_CALL_SEQUENCE error. The permissible call sequences are defined later in this chapter.

Once a transaction starts, many of the XBSA Environment variables can no longer be reset. BSA_SERVICE_HOST, NBBSA_CLIENT_HOST, NBBSA_POLICY, and NBBSA_SCHEDULE cannot be modified within a transaction. If these need to be

modified, the XBSA Application must exit the transaction, make the variable changes, and start a new transaction.

Backup transaction

A XBSA Application can create a NetBackup XBSA Object in a backup transaction. The backup transaction is defined by the first BSACreateObject() call. The BSACreateObject() function takes as input an object descriptor that has all of the XBSA attributes of the object. After the BSACreateObject() call, the object's data is passed to NetBackup in buffers using a sequence of BSASendData() calls. When all data has been sent, the object is completed with a BSAEndData() call. Multiple objects may be created in one transaction, although BSAEndData() must be called before the next BSACreateObject() is called.

The NetBackup XBSA Interface treats backup and archive transactions the same. It is up to the XBSA Application to do any extra operations that may be associated with an archival. The XBSA Application is also responsible for any other backup types such as an incremental backup. The NetBackup archive and incremental backups do not apply to the NetBackup XBSA Interface. It is also important to note that all information required to restore an object needs to be contained in the object descriptor or object data.

Within a backup transaction, query, delete, and restore operations are not allowed.

Restore transaction

The Restore transaction is similar to Backup transaction, except that the data flow is reversed. The restore transaction is defined by a call to BSAGetObject().

In order to restore an XBSA object, the NetBackup XBSA Interface needs to know the copyId of that object. The copyId can be obtained from a catalogue maintained by the XBSA Application or from a prior BSAQueryObject() call. Query operations can be mixed in with restore operations to get this data.

The BSAGetObject() call is used to initiate the restore of an object. It takes as input an object descriptor that contains the copyId of the object to be restored. Then a series of BSAGetData() calls are used to get data for the object in buffers, and the BSAEndData() call is to signal the end of getting data for the object. It is up to the XBSA Application to recreate the object being restored using the object descriptor and data. When restoring multiple objects, the XBSA Application must get all data for an object and call BSAEndData() before calling BSAGetObject() to start restoring the next object.

Within a restore transaction, it is permissible to have BSAQueryObject() and BSAGetNextQueryObject() calls. This allows the XBSA Application to intermix restore operations with BSAQueryObject() and BSAGetNextQueryObject() calls in

order to restore multiple objects within one transaction. Backup and delete operations are not allowed within a restore transaction.

It should be noted that the use of transactions for restore operations does not provide any functional benefit to the XBSA Application but is required for completeness. If a restore is aborted via a call to BSAEndTxn() or BSATerminate() before the restore has completed, the NetBackup XBSA Interface will free up the NetBackup resources but it is up to the XBSA Application to leave the object being restored in a consistent state.

Delete transaction

A XBSA Application may delete a NetBackup XBSA Object using the BSADeleteObject() call. BSADeleteObject() takes a copyId as a parameter and marks that object to be deleted. The actual delete of an object does not take place until the BSAEndTxn() call commits the transaction, so a query within a delete transaction could return an object to be deleted. If an object was backed up to a tape device, the data will not be deleted as part of this transaction. When all images on a tape have been deleted or expired, NetBackup will free the tape to be reused.

Within a delete transaction, it is permissible to embed BSAQueryObject() and BSAGetNextQueryObject() calls. This allows the XBSA Application to intermix delete operations with BSAQueryObject() and BSAGetNextQueryObject() calls in order to delete multiple objects within one transaction. Backup and restore operations are not allowed within a delete transaction.

Note: For NetBackup XBSA Version 1.1.0, BSADeleteObject() has a limitation that there can only be one object in a NetBackup image for the delete to work. This means that when the object was created, it was the only object created in the transaction. If there are multiple objects, BSADeleteObject() will return a BSA_RC_SUCCESS status, but the object will still exist.

NetBackup takes care of deleting objects via the retention period setting which is part of the configuration of a NetBackup schedule. In general, due to the way the data is stored on tape and other media, deleting individual objects has limited value.

Query transaction

A XBSA Application may query for NetBackup XBSA Objects that have been created in a query transaction. The BSAQueryObject() call is used to query the NetBackup catalogue for NetBackup XBSA Objects. Since retention of NetBackup XBSA Objects is a function of NetBackup there is no guarantee that the call to BSAQueryObject() will return any objects.

The query is based on a subset of the object descriptor attributes, contained in a query descriptor. All fields in the query descriptor must be populated and the query will search for objects that match all fields. Each of the fields does have a wildcard or 'ANY' value that can be used. But leaving a field blank will only match objects that also have blanks in that field.

The result of a query can return Object Descriptors, but never XBSA Object Data. If a query finds multiple object descriptors, BSAQueryObject() will return the first object descriptor and the remaining objects can be retrieved one at a time by using a succession of BSAGetNextQueryObject() calls.

It should be noted that the use of transactions for query operations does not provide any functional benefit to the XBSA Application but is required for completeness. And as noted in the other transaction types, queries can be embedded in restore and delete transactions.

Creating a NetBackup XBSA application

This section contains information on initiating an XBSA session, using XBSA objects, logging, running an XBSA application in a clustered environment, and hints for getting the best performance out of the NetBackup XBSA Interface.

Initiating a session

A session is initiated with a call to BSAInit(). One of the parameters of BSAInit() is the list of environment variables that is used to set up the XBSA environment between the XBSA Application and the NetBackup XBSA Interface. The only variable that is required by the NetBackup XBSA Interface is BSA_API_VERSION. BSAInit() will validate that the XBSA Application is using a supported version. Other environmental variables can be included to increase flexibility of the application or to override values from the NetBackup configuration. But if these variables are not set, there are defaults from the configuration that will be used.

Be aware that using these environment variables does not allow the XBSA Application to bypass the NetBackup configuration, only to change from the default. All hosts, policies, schedules, etc. that are used must still be defined in the NetBackup configuration in order for the transactions to work. See the *NetBackup System Administrator's Guide for UNIX, Volume I*, or *NetBackup System Administrator's Guide for Windows, Volume I*, for more information on how to configure NetBackup.

The XBSA Application should allow the XBSA environment variables to be set from run time values. These values can be obtained from parameters or from system environment variables. This will allow the maximum flexibility for the application.

See "About How to run a NetBackup XBSA application" on page 77.

Some of the XBSA environment variables must be specified in the call to BSAInit() and cannot be changed within the session. Others can be set or modified within the session which gives the XBSA Application maximum flexibility.

More information is available for the individual variables.

See See "NetBackup XBSA environment" on page 28.

Modifying XBSA environment within a session

The XBSA environment is created when the session is initiated. A couple of the variables, like BSA_API_VERSION and NBBSA_LOG_DIRECTORY, cannot be changed once the session has started. Many of the other variables can still be modified. If the XBSA Application is going to set BSA_SERVICE_HOST, NBBSA_CLIENT_HOST, NBBSA_POLICY, or NBBSA_SCHEDULE, this needs to be done outside of a transaction, either before the first transaction or between transactions.

Once within a session, the XBSA Environment can be updated with either NBBSASetEnv() or NBBSAUpdateEnv(). These are extensions to the XBSA specification. NBBSASetEnv() is used to set an individual XBSA environment variable and NBBSAUpdateEnv() updates the entire XBSA environment.

Session example

The following example sets up a session and begins a transaction. It sets up the XBSA environment, a BSA_ObjectOwner structure, and a BSA_SecurityToken. The security token is NULL because the NetBackup XBSA Interface does not use this security method. The session is initiated by a BSAInit() call that returns a BSA_Handle. This handle is then used when beginning a transaction and for all XBSA function calls within the session. Within the session, the XBSA environment is modified to change the NBBSA_CLIENT_HOST. Lastly a transaction is started.

```
BSA_Handle                  BsaHandle;
BSA_Obje ctOwner            BsaObjectOwner;
BSA_SecurityToken           *security_tokenPtr;
BSA_UInt32                  Size;
char                        *envx[3];
char                        ErrorString[512];
char                        msg[1024];
int                         status;

/ * Allocate memory for the XBSA environment variable array. */
envx[0] = malloc(40);
```

```
envx[1] = malloc(40);

/ * Populate the XBSA environment variables for this session.
  * Normally the BSA_SERVICE_HOST would not be hard coded like this but
  * would be retrieved via a parameter or environment variable.
  */
strcpy(envx[0], "BSA_API_VERSION=1.1.0");
strcpy(envx[1], "BSA_SERVICE_HOST=server_host");
envx[2] = NULL;

/ * The NetBackup XBSA Interface does not use the security token. */

security_tokenPtr = NULL;

/ * Populate the object owner structure. */

strcpy(BsaObjectOwner.bsa_ObjectOwner,"XBSA Client");
strcpy(BsaObjectOwner.app_ObjectOwner,"XBSA App");

/ * Initialize an XBSA session. */
status = BSAInit(&BsaHandle,NULL,&BsaObjectOwner,envx);
if (status != BSA_RC_SUCCESS) {
   Size = 512;
   NBBSAGetErrorString(status, &Size, ErrString);
   printf("ERROR: BSAInit failed with error: %s\n", ErrString);
   exit(status);
}

/ * Set the hostname of the client for the next transaction.  */
NBBSASetEnv(BsaHandle, "NBBSA_CLIENT_HOST", "client_host");

/ * Begin a transaction.  If it fails, terminate the session. */
status = BSABeginTxn(BsaHandle);
if (status != BSA_RC_SUCCESS) {
   Size = 512;
   NBBSAGetErrorString(status, &Size, ErrorString);
   sprintf(msg, "ERROR: BSABeginTxn failed with error: %s",
       ErrorString);
   NBBSALogMsg(BsaHandle, MSERROR, msg, "Backup");
   BSATerminate(BsaHandle);
   exit(status);
}
```

Backup - creating an object

Once the application has started a transaction, it can start a backup. A backup transaction is identified by the first BSACreateObject() call. BSACreateObject() will start the process of backing up an object. Once the object has been created, multiple BSASendData() calls are be used to send the data associated with an object. This object is then completed with a BSAEndData() call.

The ability to pass data in buffers allows an XBSA Application to use any buffering technique that is appropriate to ensure consistency or to improve performance. When data is passed in buffers, all the data for one object must be passed, in the proper sequence, before any other operation is started.

Creating an object

An object descriptor defines an XBSA object. It is up to the XBSA Application to define the attributes of the object such that the application will know how to restore the object. For example, if the XBSA Application wants to implement an incremental type of backup, enough information will need to be kept in the object descriptor to identify if the object is a full or incremental and any other information that will be required to restore the object.

The following fields of an object descriptor are user-defined and need to be defined by the XBSA Application before the descriptor is passed to BSACreateObject().

See See "Object descriptors" on page 22. for more definition of the BSA_ObjectDescriptor.

The fields that are defined as strings can be empty strings, except for the pathName, which must have a valid path. The fields that are enumerations cannot have the ANY value. The estimatedSize field must have a value greater than zero if the object will have data and zero if there will be no data. While it is good practice to have the estimated size field be as accurate as possible, it does not affect how NetBackup will store the object.

The following are the required BSA.ObjectDescriptor fields:

```
objectOwner
   bsa_objectOwner
   app_objectOwner
objectName
   pathName
   objectSpaceName
copyType
resourceType
objectType
objectDescription
```

```
estimatedSize
objectInfo
```

The NetBackup XBSA Interface will populate the other fields in the object descriptor.

The other structure that is required before creating an object is the BSA_DataBlock32 structure. The structure does not need to be populated because BSACreateObject() will populate select fields with values that define how the data needs to be passed in buffers.

See "Buffers" on page 25..

Those are the two parameters to BSACreateObject(). The BSACreateObject() function will create the object and prepare the NetBackup to be able to accept data. This includes mounting a tape if that is required. When BSACreateObject() has successfully created the object and returns, the object descriptor will have the copyId field populated. This is the unique identifier that is associated with this object. If the XBSA Application is going to keep any information about an object in an application catalog, this copyId should be a key value. It can be used to restore or delete this object.

There are four environmental variables that are created during BSACreateObject(). These are NBBSA_CLIENT_READ_TIMEOUT, NBBSA_MEDIA_MOUNT_TIMEOUT, NBBSA_MULTIPLEXING, and NBBSA_SERVER_BUFFSIZE. These variables are part of the NetBackup configuration and can be used to determine if the XBSA application will be successful. The NBBSA_CLIENT_READ_TIMEOUT and NBBSA_MEDIA_MOUNT_TIMEOUT values can be reset by the XBSA application if it knows it needs to override the default NetBackup configuration.

NBBSA_CLIENT_READ_TIMEOUT is the amount of time, in seconds, the NetBackup server will wait for data to be received. If the time between when the NetBackup server starts the backup and the time the transmission of data starts exceeds this timeout value, the backup job will fail. This is to ensure that a hung or failed process on the client does not cause the job to wait, and take up resources, indefinitely. If the XBSA Application knows it will take longer than this to prepare the data to be sent, this value should be reset to a higher value.

NBBSA_MEDIA_MOUNT_TIMEOUT is the amount of time the NetBackup client will wait for the media to be mounted. If the time between when the NetBackup server starts the backup and the time the media is mounted exceeds this timeout value, the XBSA Interface will return a fail condition.

NBBSA_MULTIPLEXING is the number of streams that can be accepted by NetBackup. This value cannot be changed but if the XBSA Application is processing multiple streams, it should be evaluated to make sure that NetBackup will accept all streams that are being sent.

NBBSA_SERVER_BUFFSIZE is the size configured for NET_BUFF_SZ. This value cannot be changed but, if the XBSA Application has the ability to modify the size of the buffers it uses, these could be modified to enhance performance of the transfer of data.

If everything is OK so far, data can be sent to the NetBackup XBSA Interface via buffers by BSASendData(). The buffers are defined by the BSA_DataBlock32 structure. The key fields to set are the numBytes, which contains the number of bytes being sent, bufferLen, which contains the length of the buffer in bytes, and bufferPtr, which is a pointer to the buffer. The number of bytes must equal the buffer length except for the last buffer, which can be only partially full. BSASendData() can be called any number of times to pass all the data from an object.

Once all data has been sent, BSAEndData() must be called to signal to the NetBackup XBSA Interface that the object is complete.

If multiple objects are to be created, this whole process can be repeated multiple times. The most efficient way to create multiple objects is to repeat this within one transaction. It is also possible to create multiple objects by creating one object per transaction and doing multiple transactions.

Once all objects for a transaction have been created, the transaction is completed with BSAEndTxn(). BSAEndTxn() can either commit or abort the transaction. If the transaction is aborted, all objects that were created in the transaction are not saved. If the transaction is committed, the object(s) are saved in the NetBackup catalog and can at a future point be restored. The BSATerminate() function also acts as an abort to the transaction.

NetBackup object ownership

Default behavior

When the NetBackup XBSA interface is used to create an object, by default the owner of the object will be the login user of the process that created the object. The default group of the object will also be the login user, not the primary group of the login user, but the exact same name as the login user name. The permissions of the file will be set to 600, or

‘rw- - - - - - -’, which is read/write for owner and no access permissions for anyone else. This requires that the user restoring an object be an administrator or the same user that created the object. The XBSA objectOwner fields are saved in the NetBackup catalog with the object, but they are kept as attributes of the object and are not used for security purposes.

Ownership options

Using the XBSA environmental variables NBBSA_USE_OBJECT_OWNER, NBBSA_USE_OBJECT_GROUP, NBBSA_OBJECT_OWNER, and NBBSA_GROUP_OWNER, an agent can change the default owner. These variables allow the XBSA agent to be able to specify who owns the objects.

Note: Specifying object ownership only works when creating objects using BSACreateObject(). Accessing the objects via BSAQueryObject() and BSAGetObject() is dependent on the login process having permissions to access the objects. So if user_Y creates an object with an object owner of user_X, then user_X or an administrator (root) can access and restore the object, but user_Y cannot.

Object owner

To specify the owner of an object, the XBSA environment variable NBBSA_USE_OBJECT_OWNER needs to be set. There are 4 values that this variable can be set to. These values are defined in `nbbsa.h`.

```
/*
 * XBSA values to use to define how to specify NetBackup object ownership
 */
#define VxLOGIN_USER  0 /* Default, owner/group field is set to the login user */
#define VxLOGIN_GROUP 1 /* group field is set to the primary group of the login user */
#define VxBSA_OWNER   2 /* owner/group field is set to                              \
objectDesctiptor->objectOwner.bsa_ObjectOwner  */
#define VxAPP_OWNER   3 /* owner/group field is set to                              \
objectDesctiptor->objectOwner.app_ObjectOwner  */
#define VxENV_OWNER   4 /* owner/group field is set to value of                     \
NBBSA_OBJECT_OWNER/NBBSA_OBJECT_GROUP */
```

VxLOGIN_USER is the default behavior that you would get if the NBBSA_USE_OBJECT_OWNER variable wasn't set.

VxLOGIN_GROUP does not apply to object ownership.

VxBSA_OWNER will set the object owner to the value stored in the objectDescriptor field objectOwner.bsa_ObjectOwner. The value in the bsa_ObjectOwner field will need to be a valid username without any spaces in the name. The value in objectOwner.bsa_ObjectOwner will still be stored as an attribute of the object and a query will need to correctly specify this field in the query descriptor to successfully find the object.

VxAPP_OWNER will set the object owner to the value stored in the objectDescriptor field objectOwner.app_ObjectOwner. The value in the app_ObjectOwner field will need to be a valid username without any spaces in the name. The value in objectOwner.app_ObjectOwner will still be stored as an attribute of the object and

a query will need to correctly specify this field in the query descriptor to successfully find the object.

VxENV_OWNER will set the object owner to the value of the XBSA environmental variable NBBSA_OBJECT_OWNER. The value stored in the NBBSA_OBJECT_OWNER will need to be a valid username without any spaces in the name.

The variables NBBSA_USE_OBJECT_OWNER and NBBSA_OBJECT_OWNER can be changed within a transaction so that an XBSA agent can set different ownership of each object in a transaction if it so desires.

Object group

An XBSA agent can also change the group ownership of an object. When the group ownership is set via one of these options, other than the default, the permissions on the object are set to 660, or 'rw - rw- - - -', which is read/write for owner and group. This allows any user in the specified group to access and restore the object.

To specify the group of an object, the XBSA environment variable NBBSA_USE_OBJECT_GROUP needs to be set. There are 5 values that this variable can be set to. These values are defined in nbbsa.h.

```
/*
 * XBSA values to use to define how to specify NetBackup object ownership
 */
#define VxLOGIN_USER 0 /* Default, owner/group field is set to the login user  */
#define VxLOGIN_GROUP 1 /* group field is set to the primary group of the login user */
#define VxBSA_OWNER 2 /* owner/group field is set to                                  \
objectDesctiptor->objectOwner.bsa_ObjectOwner  */
#define VxAPP_OWNER 3 /* owner/group field is set to                                  \
objectDesctiptor->objectOwner.app_ObjectOwner  */
#define VxENV_OWNER 4 /* owner/group field is set to value of                         \
NBBSA_OBJECT_OWNER/NBBSA_OBJECT_GROUP */
```

VxLOGIN_USER is the default behavior that you would get if the NBBSA_USE_OBJECT_GROUP variable wasn't set. The group name will be the same name as the owner field, whether that is the login user or a user name defined by one of the other options, and the permissions of the object will be 600, owner read/write only.

VxLOGIN_GROUP will set the group field to the primary group of the login user.

VxBSA_OWNER will set the object group to the value stored in the objectDescriptor field objectOwner.bsa_ObjectOwner. The value in the bsa_ObjectOwner field will need to be a valid username without any spaces in the name. The value in objectOwner.bsa_ObjectOwner will still be stored as an attribute of the object and

a query will need to correctly specify this field in the query descriptor to successfully find the object.

VxAPP_OWNER will set the object group to the value stored in the objectDescriptor field objectOwner.app_ObjectOwner. The value in the app_ObjectOwner field will need to be a valid username without any spaces in the name. The value in objectOwner.app_ObjectOwner will still be stored as an attribute of the object and a query will need to correctly specify this field in the query descriptor to successfully find the object.

VxENV_OWNER will set the object group to the value of the XBSA environmental variable NBBSA_OBJECT_GROUP. The value stored in the NBBSA_OBJECT_GROUP will need to be a valid username without any spaces in the name.

The variables NBBSA_USE_OBJECT_GROUP and NBBSA_OBJECT_GROUP can be changed within a transaction so that an XBSA agent can set different group ownership of each object in a transaction if it so desires.

Creating an empty object

It is acceptable to create an XBSA object without any associated data. This is created in much the same way as an object with data with two differences. The estimatedSize.left and estimatedSize.right fields need to be zero so the NetBackup XBSA Interface knows that the object is going to be empty. After the BSACreateObject() call, the XBSA Application calls BSAEndData() to end the object. If estimatedSize is set to zero and BSASendData() is called, this will result in an error.

Backup example

The following example goes through the process of creating an object. It is assumed the transaction has already been started). The BSA_ObjectDescriptor is populated with the values that are associated with the object. Then the DataBlock32 structure is created to receive any restrictions put on the data by the NetBackup Interface. BSACreateObject() is then called to create the object and start the backup process. Once the object is created, this example sends one buffer of data with the BSASendData() call. After the last BSASendData() call, the object is completed with a BSAEndTxn(), which commits the object.

This highly simplistic example only creates one object and only sends one buffer of data. In general, objects will take multiple buffers and a transaction can create multiple objects.

```
BSA_Handle                      BsaHandle;
BSA_ObjectOwner                 BsaObjectOwner;
BSA_SecurityToken               *security_tokenPtr;
```

```
        BSA_DataBlock32           *data_block;
BSA_ObjectDescriptor              *object_desc;
BSA_UInt32                        DataBuffSz;
BSA_UInt32                        Size;
char                              *envx[5];
char                              DataBuff[512];
char                              ErrorString[512];
char                              msg[1024];
int                               status;
.
.
BSAInit(&BsaHandle, security_tokenPtr, &BsaObjectOwner, envx);
.
.
BSABeginTxn(BsaHandle);

/ * Populate object descriptor of the first object to be backed up. */

object_desc = (BSA_ObjectDescriptor *)malloc(sizeof(BSA_ObjectDescriptor));

strcpy(object_desc->objectOwner.bsa_ObjectOwner, "XBSA Client");
strcpy(object_desc->objectOwner.app_ObjectOwner, "XBSA App");
strcpy(object_desc->objectName.pathName, "/xbsa/sample/object1");
strcpy(object_desc->objectName.objectSpaceName, "");
strcpy(object_desc->resourceType, "Sample");
strcpy(object_desc->objectDescription,"Sample description of Obj 1");
strcpy(object_desc->objectInfo,"Object 1");
object_desc->copyType = BSA_CopyType_BACKUP;
object_desc->estimatedSize.left = 0;
object_desc->estimatedSize.right = 100;
object_desc->objectType = BSA_ObjectType_FILE;

/ *  Initialize the BSA_DataBlock32 structure. */

data_block = (BSA_DataBlock32 *)malloc(sizeof(BSA_DataBlock32));
memset(data_block, 0x00, sizeof(BSA_DataBlock32));
/ * Create sample object. If object cannot be created, terminate session. */

status = BSACreateObject(BsaHandle, object_desc, data_block);
if (status == BSA_RC_SUCCESS) {
    printf("copyId: %d - %d\n", object_desc->copyId.left, object_desc->copyId.right);
} else {
    Size = 512;
```

```
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSACreateObject failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Backup");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

/ * For the purposes of this sample, we will assume that the data in the *
  * DataBuff buffer has been populated from reading the data from the object *
  * being backed up. */

strcpy(DataBuff, "This is the sample data that is contained in the sample object that
is being backed up for the purposes of showing how data can be backed up and
restored.");
DataBuffSz = strlen(DataBuff);

/ * BSACreateObject sets values in the BSA_DataBlock32 to indicate    *
  * header and trailer requirements. The NetBackup implementation has *
  * no such requirements and are not checked here.  Set the other     *
  * fields of the data_block for the BSASendData call.              */

data_block->bufferLen = 512;
data_block->bufferPtr = DataBuff;
data_block->numBytes = DataBuffSz;

/ * Send the data to be backed up.  In normal implementations, BSASendData    *
  * would be in a loop reading the data into the buffer and then sending it   *
  * until all the data of the object has been sent.                        */

status = BSASendData(BsaHandle, data_block);
if (status == BSA_RC_SUCCESS) {
    printf("Bytes backed up: %d\n", data_block->numBytes);
} else {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSASendData failed with error: %s\n", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Backup");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}
```

```
/ *  All data has been sent for the object.  */

status = BSAEndData(BsaHandle);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndData failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Backup");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

/ *  End the backup transaction and commit the object.  */

status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndTxn failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Backup");
    BSATerminate(BsaHandle);
    exit(status);
}
```

Query - finding an object descriptor

An XBSA Application may query the NetBackup XBSA Interface for XBSA objects that have been created. The BSAQueryObject() call is used to query the NetBackup catalog for these objects. The query is based on a subset of the object descriptor attributes, contained in a query descriptor. If the result of the query is multiple object descriptors, BSAQueryObject() will return the first (most recent) object and the rest can be retrieved one object descriptor at a time by using a succession of BSAGetNextQueryObject() calls.

Querying for an object

When querying for an object, the object attributes that the XBSA Application is querying for are contained in a query descriptor. This query descriptor is made up of strings and enumerations. They will be evaluated against the objects stored in the NetBackup catalog for objects that match all fields. Each field of the query descriptor must be populated. If a string field is populated with an empty string or NULL, it will only match objects that also have an empty string for that field.

Wildcards and 'ANY' enumerations allow the XBSA application to search for objects that have some fields that are unknown.

There are two fields that are not part of the XBSA specifications but can be very useful. The createTime_from and createTime_to fields limit the search to the time period between these dates. These are optional fields, the default is to search all objects, but can greatly speed up the search when the NetBackup catalog is very large.

When doing the query, the XBSA application will only return objects that are owned by login user running the query, unless that user is a root admin. NetBackup XBSA Version 1.1.0 uses the login user as the object owner. The objectOwner field is considered an attribute and is not used for security.

The query, by default, will also only return objects that were created on the hostname from which the query is being run. If the XBSA Application needs to find an object that was created from a different host, the NBBSA_CLIENT_HOST environment variable must be set to the hostname from which the object was created. This variable can only be set before a transaction begins. If the application is looking for objects from multiple hosts, the application will need to do queries in separate transactions.

Query example

Here is a simple example of a query. It starts with populating a query descriptor, which identifies what objects are being searched for. Then it makes the initial query

```
BSA_Handle                  BsaHandle;
BSA_ObjectOwner             BsaObjectOwner;
BSA_SecurityToken           *security_tokenPtr;
BSA_QueryDescriptor         *query_desc;
BSA_ObjectDescriptor        *object_desc;
BSA_UInt32                  Size;
char                        *envx[3];
char                        ErrorString[512];
char                        msg[1024];
int                         status;
.
.
BSAInit(&BsaHandle, security_tokenPtr, &BsaObjectOwner, envx);
.
.
BSABeginTxn(BsaHandle);
```

```
/ *  Populate the query descriptor of the object to be searched for. */

query_desc = (BSA_QueryDescriptor *)malloc(sizeof(BSA_QueryDescriptor));

query_desc->copyType = BSA_CopyType_BACKUP;
query_desc->objectType = BSA_ObjectType_FILE;
query_desc->objectStatus = BSA_ObjectStatus_ANY;
strcpy(query_desc->objectOwner.bsa_ObjectOwner, "XBSA Client");
strcpy(query_desc->objectOwner.app_ObjectOwner, "XBSA App");
strcpy(query_desc->objectName.pathName, "/xbsa/sample/object1");
strcpy(query_desc->objectName.objectSpaceName, "");

object_desc = (BSA_ObjectDescriptor *)malloc(sizeof(BSA_ObjectDescriptor));

/ * Begin searching for objects matching the query criteria.  BSAQueryObject()    *
  * returns the first (most recent) object found.                                 */

status = BSAQueryObject(BsaHandle, query_desc, object_desc);
if (status == BSA_RC_SUCCESS) {
    printf("copyId: %d - %d\n", object_desc->copyId.left, object_desc->copyId.right);
} else if (status == BSA_RC_NO_MATCH) {
    sprintf(msg, "WARNING: BSAQueryObject() did not find an object matching the
       query");
    NBBSALogMsg(BsaHandle, MSWARNING, msg, "Query");
    BSATerminate(BsaHandle);
    exit(status);
} else {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAQueryObject() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Query");
    BSATerminate(BsaHandle);
    exit(status);
}

/ * Continue searching for other objects which match the query criteria.  *
  * BSAGetNextQueryObject() should return BSA_RC_NO_MORE_DATA when there  *
  * are not more objects.                                                 */

while ((status = BSAGetNextQueryObject(BsaHandle, object_desc)) == BSA_RC_SUCCESS) {
    printf("CopyId: %d.%d\n", object_desc->copyId.left, object_desc->copyId.right);
}
if (status != BSA_RC_NO_MORE_DATA) {
```

```
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAGetNextQueryObject() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Query");
    BSATerminate(BsaHandle);
    exit(status);
}

/ * End the query transaction.  BSA_Vote_COMMIT and BSA_Vote_ABORT are *
  * equivalent as there is nothing to commit or abort.                 */

status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndTxn() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Query");
    BSATerminate(BsaHandle);
    exit(status);
}
```

Restore - retrieving an object's data

Another type of transaction is a restore transaction. A restore transaction is identified by the first BSAGetObject() call. A difference from a backup transaction is that there can also be BSAQueryObject() calls within a restore transaction, which is useful to get the object descriptor of the object the XBSA Application is restoring. BSAGetObject() will start the process of retrieving an object. Once the object has been retrieved, multiple BSAGetData() calls are be used to retrieve the data associated with an object. The last BSAGetData() call will return BSA_NO_MORE_DATA that will signal that the NetBackup XBSA Interface has completed sending the data. The BSAEndData() call will then release all resources.

Restoring an object

When restoring an XBSA object, the login user must be the owner of the XBSA object or a root admin. (The owner of an object is the login user of the process that created the object.) If a different user tries to restore the object, the NetBackup XBSA Interface will return a BSA_RC_OBJECT_NOT_FOUND error. This error could also be returned if the host on which the restore is being done is different from the host which backed up the object.

See "Redirected restore to a different client" on page 57.

The XBSA Application is responsible for recreating the object. The NetBackup XBSA Interface sends a stream of data to the XBSA Application. It is up to the XBSA Application to ensure the correct permissions exist for restoring the object, recreating all attributes, etc. If any of these attributes are stored in the object descriptor of the XBSA object, the object descriptor needs to be retrieved with a BSAQueryObject() call. The call to BSAGetObject() does not populate the object attributes.

To restore an XBSA object, the NetBackup XBSA Interface needs to have an object descriptor that contains the copyId of the object being restored. This copyId can be obtained from either a query process or from information stored by the XBSA Application. It is permissible to mix query operations in a restore transaction.

The other structure that is required before restoring an object is the BSA_DataBlock32 structure. The structure does not need to be populated as BSAGetObject() will populate select fields with values that define how the data buffers will be used.

See "Buffers" on page 25.

The restore is initiated with a call to BSAGetObject() with this object descriptor and data block as parameters. This function starts the process of retrieving the object. If BSAGetObject() returns with good status, BSAGetData() can retrieve the object data from the NetBackup XBSA Interface via buffers. The buffers are defined by the BSA_DataBlock32 structure. It is the responsibility of the XBSA Application to allocate the buffers. BSAGetObject() will fill the buffers with data and set the numBytes field of the BSA_DataBlock32 with the number of bytes in the buffer. When the last buffer of data for the object has been passed, BSAGetObject() will return BSA_NO_MORE_DATA. BSAEndData() should then be called to signal to the NetBackup XBSA Interface that the object is restored and that it can free up the resources. The NetBackup XBSA Interface requires that all data for an object is retrieved or the return status of the NetBackup server would be an error status. This will not affect the XBSA Application, but may impact how a user of the application interprets the results of the restore.

After the object(s) have been restored, the transaction should be closed. From the NetBackup XBSA Interface point of view, a committed or aborted transactions are handled the same, as there is nothing for NetBackup to commit.

Redirected restore to a different client

One specific type of restore that deserves special notice is what is considered a redirected restore to a different client. An XBSA object is stored within NetBackup with a specific client from which it was backed up. The default is to assume that the object is being restored to the same client. If the hostname that is initiating

the restore is different from the hostname on which the object was backed up, the NBBSA_CLIENT_HOST environment variable needs to be set.

The NBBSA_CLIENT_HOST must be set, before entering the transaction, to the hostname on which the object was backed up. If this variable has not been specified, the NetBackup XBSA Interface will not be able to find the object.

Restore example

Here is an example of a restore. It assumes that the object descriptor has been populated with the copyId of the object either from a query or the XBSA application having stored this information.

```
BSA_Handle                      BsaHandle;
BSA_ObjectOwner                 BsaObjectOwner;
BSA_SecurityToken               *security_tokenPtr;
BSA_DataBlock32                 *data_block;
BSA_UInt32                      EnvBufSz = 512;
BSA_ObjectDescriptor            *object_desc;
BSA_QueryDescriptor             *query_desc;
BSA_UInt32                      Size;
char                            *envx[3];
char                            EnvBuf[512];
char                            ErrorString[512];
char                            msg[1024];
char                            *restore_location;
int                             total_bytes = 0;
int                             status;
.
.
BSAInit(&BsaHandle, security_tokenPtr, &BsaObjectOwner, envx);
.
.
BSABeginTxn(BsaHandle);

/ *  Get the object.  */

data_block = (BSA_DataBlock32 *)malloc(sizeof(BSA_DataBlock32));

status = BSAGetObject(BsaHandle, object_desc, data_block);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAQueryObject() failed with error: %s", ErrorString);
```

```
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Restore");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

/ * The application is responsible for recreating the file or other object   *
/ * type that is being restored using the information that is stored in the  *
/ * object_descriptor.  This sample prints the results to the screen.        */

restore_location = (char *)malloc((EnvBufSz + 1) * sizeof(char));
memset(restore_location, 0x00, EnvBufSz + 1);

/ * Initialize the data_block structure.  */

data_block->bufferLen = EnvBufSz;
data_block->bufferPtr = EnvBuf;
memset(data_block->bufferPtr, 0x00, EnvBufSz);

/ * Read data until the end of data. */

while ((status = BSAGetData(BsaHandle, data_block)) == BSA_RC_SUCCESS) {

    / * Move the retrieved data to where it is to be restored to and  *
      * reset the data_block buffer.                                   */

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;

    printf("%s", restore_location);

    memset(restore_location, 0x00, EnvBufSz + 1);
    memset(data_block->bufferPtr, 0x00, EnvBufSz);
}
if (status == BSA_RC_NO_MORE_DATA) {

    / * The last BSAGetData() that returns BSA_RC_NO_MORE_DATA may have data  *
      * in the buffer.                                                        */

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;

    printf("%s\n", restore_location);
```

```
    printf("Total bytes retrieved: %d\n", total_bytes);
} else {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAGetData() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Restore");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

/ * Done retrieving data. */

status = BSAEndData(BsaHandle);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndData() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Restore");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

/ * End the restore transaction.  BSA_Vote_COMMIT and BSA_Vote_ABORT are         *
/ * equivalent as there is nothing to commit or abort for a restore transaction. */

status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndTxn() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Restore");
    BSATerminate(BsaHandle);
    exit(status);
}
```

Multiple object restore

If multiple objects are going to be restored in one session or transaction, the XBSA agent should consider using the NBBSAGetMultipleObjects function call. This is a NetBackup extension to the XBSA interface to optimize the retrieval of objects

in a NetBackup environment. This is especially useful when retrieving many small objects.

The reason this provides a performance improvement is that each NetBackup restore operation creates a NetBackup job, which acquires resources and then frees them up when the job is complete. Each BSAGetObject call translates into one NetBackup job. The initial time required to start a NetBackup job and establish communication are minimal, especially when compared to the time to transfer large amounts of data. But if the objects are small and numerous, this overhead per object will be noticeable. It is also possible on heavily loaded NetBackup systems that successive BSAGetObject calls may get queued up behind other NetBackup jobs and resource requests. Any of these could cause performance issues if the separate objects are really all part of one restore.

To address this behavior of NetBackup, we have added a multiple object restore interface to the XBSA interface. This is an extension of the XBSA specification to enhance performance for NetBackup XBSA Applications. The use of this interface is not required and does not provide functionality on objects that is not available through the interfaces defined by XBSA.

Requirements

- In order to do a multiple object restore, the XBSA Application needs to have created the objects in ways that will allow this and there are restrictions on how the objects can be retrieved.
- All the objects to be restored within a multiple object restore must be part of the same NetBackup image, which means that the objects were created in one transaction. This can be verified by checking that each object being restored has the same copyId.left.
- The objects must be retrieved in the same order that they were created. Some objects in the image can be skipped, but the objects being retrieved must be ordered in a way that will not cause the media to have to position backwards. The ordering of objects can be determined by verifying that the copyId.right for each object is in ascending order.
- While not all objects in an image need to be retrieved, all objects in the list created by NBBSAAddToMultiObjectRestoreList must be retrieved in the order in which they are on the list. Objects cannot be skipped or added.
- Each object in the list will be retrieved with BSAGetObject followed by successive BSAGetData calls to retrieve all the data. All data for an object must be retrieved before moving on to the next object.

Functions and use

There are three new functions provided as part of the XBSA interface that can be used to do multiple object restores.

- NBBSAAddToMultiObjectRestoreList takes an object descriptor and it to a descriptor list. This function is called for each object that is to be retrieved as part of one restore job. It is highly recommended to use this function to create the list because it allows the XBSA interface to be in charge of memory allocation and deletion.
- NBBSAGetMultipleObjects starts the multiple object restore job. It takes the list of descriptors built by NBBSAAddToMultiObjectRestoreList and submits a request to restore all objects.
- NBBSAEndGetMultipleObjects ends the multiple object restore job. This function cleans up the memory from the object list and allows the application to COMMIT or ABORT the restore, which has no real effect on the data.

The process is very similar to the single object restores. First, all objectDescriptors to be retrieved are added to a list using the NBBSAAddToMultiObjectRestoreList. The objectDescriptors can be generated from BSAQueryObject or populated by the application. Once the list is ready, a call to NBBSAGetMultipleObjects will initiate the restore process. Then, each object is retrieved using the standard BSAGetObject, BSAGetData, and BSAEndData function calls. The difference is that BSAGetObject knows it is part of a larger restore job. After all objects have been retrieved, NBBSAEndGetMultipleObjects is called to end the restore process. The transaction can then be ended. If an object is skipped or not all data is retrieved, the entire job will fail.

Multiple object restore example

Here is an example of a multiple object restore. Examples of BSAQueryObject and BSAGetObject are included elsewhere in this document, so this example bypasses some of the error handling associated with those calls.

```
BSA_Handle                  BsaHandle;
BSA_ObjectOwner             BsaObjectOwner;
BSA_SecurityToken           *security_tokenPtr;
BSA_DataBlock32             *data_block;
BSA_QueryDescriptor         *query_desc;
BSA_ObjectDescriptor        *object_desc;
BSA_ObjectDescriptor        *object_desc_current;
NBBSA_DESCRIPT_LIST         *object_list = NULL;
NBBSA_DESCRIPT_LIST         *object_list_current;
BSA_UInt32                  EnvBufSz = 512;
BSA_UInt32                  Size;
char                        *envx[3];
char                        EnvBuf[512];
char                        ErrorString[512];
```

```
char                            msg[1024];
char                            *restore_location;
int                             total_bytes = 0;
int                             status;
.
.
BSAInit(&BsaHandle, security_tokenPtr, &BsaObjectOwner, envx);
.
.
BSABeginTxn(BsaHandle);

/* Populate the query descriptor of the object to be searched for. */

query_desc = (BSA_QueryDescriptor *)malloc(sizeof(BSA_QueryDescriptor));
object_desc = (BSA_ObjectDescriptor *)malloc(sizeof(BSA_ObjectDescriptor));
data_block = (BSA_DataBlock32 *)malloc(sizeof(BSA_DataBlock32));

query_desc->copyType = BSA_CopyType_BACKUP;
query_desc->objectType = BSA_ObjectType_FILE;
query_desc->objectStatus = BSA_ObjectStatus_MOST_RECENT;
strcpy(query_desc->objectOwner.bsa_ObjectOwner, "BSA Client");
strcpy(query_desc->objectOwner.app_ObjectOwner, "BSA App");
strcpy(query_desc->objectName.pathName, "/xbsa/sample/object1");
strcpy(query_desc->objectName.objectSpaceName, "");

/* Search for an object matching the query criteria.  */

status = BSAQueryObject(BsaHandle, query_desc, object_desc);
if (status != BSA_RC_SUCCESS) {
   /* handle error condition */
}

/* Start building the objectList by adding the object descriptor to the list.  */

status = NBBSAAddToMultiObjectRestoreList(BsaHandle, &object_list, object_desc);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: NBBSAAddToMultiObjectRestoreList() failed with error: %s",
        ErrorString);
    NBBSALogMsg(BsaHandle, ERROR, msg, " Multiple Object Restore");
    BSATerminate(BsaHandle);
    exit(status);
```

```
    }

/* Search for a second object.  */

strcpy(query_desc->objectName.pathName, "/xbsa/sample/object2");
status = BSAQueryObject(BsaHandle, query_desc, object_desc);
if (status != BSA_RC_SUCCESS) {
   /* handle error condition */
}

/* Add the second object descriptor to the objectList.  */

status = NBBSAAddToMultiObjectRestoreList(BsaHandle, &object_list, object_desc);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: NBBSAAddToMultiObjectRestoreList() failed with error: %s",
        ErrorString);
    NBBSALogMsg(BsaHandle, ERROR, msg, " Multiple Object Restore");
    BSATerminate(BsaHandle);
    exit(status);
}

/* Start the multiple object restore by passing in the object list. The object list
 * will be evaluated and the restore job will be started.
 */

status = NBBSAGetMultipleObjects(BsaHandle, object_list);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: NBBSAGetMultipleObjects () failed with error: %s",
        ErrorString);
    NBBSALogMsg(BsaHandle, ERROR, msg, "Multiple Object Restore");
    BSATerminate(BsaHandle);
    exit(status);
}

/* Create a pointer to the object list in order to keep track of the current object
 * being restored.  A list created by the application could also be used.
 * Point the object descriptor at the first object
 */
object_list_current = object_list;
```

```
object_desc_current = object_list_current->Descriptor;

/*  Get the first object.  */

status = BSAGetObject(BsaHandle, object_desc_current, data_block);
if (status != BSA_RC_SUCCESS) {
    /* handle error condition */
}

restore_location = (char *)malloc((EnvBufSz + 1) * sizeof(char));
memset(restore_location, 0x00, EnvBufSz + 1);

data_block->bufferLen = EnvBufSz;
data_block->bufferPtr = EnvBuf;
memset(data_block->bufferPtr, 0x00, EnvBufSz);

/* Read data until the end of data. */

while ((status = BSAGetData(BsaHandle, data_block)) == BSA_RC_SUCCESS) {

    /* Move the retrieved data to where it is to be restored to and  *
     * reset the data_block buffer.                                  */

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;

    memset(restore_location, 0x00, EnvBufSz + 1);
    memset(data_block->bufferPtr, 0x00, EnvBufSz);
}
if (status == BSA_RC_NO_MORE_DATA) {

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;

    printf("Total bytes retrieved: %d\n", total_bytes);
} else {
    /* handle error condition */
}

/* Done retrieving data for the first object. */

status = BSAEndData(BsaHandle);
if (status != BSA_RC_SUCCESS) {
```

```
    /* handle error condition */
}

/* Set the object descriptor to the next object in the list.  */
object_list_current = object_list_current->next;
object_desc_current = object_list_current->Descriptor;

if (object_desc_current == NULL) {
   /* handle error condition */
}

/*  Get the next object.  */

status = BSAGetObject(BsaHandle, object_desc_current, data_block);
if (status != BSA_RC_SUCCESS) {
    /* handle error condition */
}

restore_location = (char *)malloc((EnvBufSz + 1) * sizeof(char));
memset(restore_location, 0x00, EnvBufSz + 1);

data_block->bufferLen = EnvBufSz;
data_block->bufferPtr = EnvBuf;
memset(data_block->bufferPtr, 0x00, EnvBufSz);

/* Read data until the end of data. */

while ((status = BSAGetData(BsaHandle, data_block)) == BSA_RC_SUCCESS) {

    /* Move the retrieved data to where it is to be restored to and  *
     * reset the data_block buffer.                                  */

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;

    memset(restore_location, 0x00, EnvBufSz + 1);
    memset(data_block->bufferPtr, 0x00, EnvBufSz);
}
if (status == BSA_RC_NO_MORE_DATA) {

    memcpy(restore_location, data_block->bufferPtr, data_block->numBytes);
    total_bytes += data_block->numBytes;
```

```
    printf("Total bytes retrieved: %d\n", total_bytes);
} else {
    /* handle error condition */
}

/* Done retrieving data for the second object. */

status = BSAEndData(BsaHandle);
if (status != BSA_RC_SUCCESS) {
    /* handle error condition */
}

/* End the multiple object restore transaction. Set any references to objects
 * in the object list to NULL as the memory associated to the list has been freed.
 */

status = NBBSAEndGetMultipleObjects(BsaHandle, BSA_Vote_COMMIT, object_list);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: NBBSAEndGetMultipleObjects() failed with error: %s",
        ErrorString);
    NBBSALogMsg(BsaHandle, ERROR, msg, "Multiple Object Restore");
    BSATerminate(BsaHandle);
    exit(status);
}
object_list_current = NULL;
object_desc_current = NULL;

/* End the restore transaction. BSA_Vote_COMMIT and BSA_Vote_ABORT are
 * equivalent as there is nothing to commit or abort for a restore transaction.
 */

status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndTxn() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, ERROR, msg, " Multiple Object Restore");
    BSATerminate(BsaHandle);
    exit(status);
}
```

Delete - deleting an object

Deleting a NetBackup XBSA Object is done with the BSADeleteObject() function. BSADeleteObject() will not always delete the object specified, even if it return a success status. The only objects that can be deleted are objects in which there was only one object created per transaction. Also note that it is possible for a deleted object to show up again as delete only removes the entry from the NetBackup catalog and the objects are not deleted from the tape media they are on. NetBackup allows media to be imported to recreate all images from that media, which could recreate an object that was deleted.

Based on those limitations, the BSADeleteObject() function is pretty straightforward. It takes a copyId as its parameter and deletes this object. Multiple objects can be deleted in one transaction and it is permissible to have query operations within a delete transaction. The object is not deleted until the transaction is committed so these query operations in a delete transaction could return a deleted object.

Delete example

This delete example is very simple. It assumes that the copyId has been populated from a previous query or from information stored by the XBSA Application. It then deletes one object and commits the transaction that does the delete of the object.

```
BSA_Handle                      BsaHandle;
BSA_ObjectOwner                 BsaObjectOwner;
BSA_SecurityToken               *security_tokenPtr;
BSA_ObjectDescriptor            *object_desc;
BSA_QueryDescriptor             *query_desc;
BSA_UInt32                      Size;
char                            *envx[3];
char                            ErrorString[512];
char                            msg[1024];
int                             status;
.
.
BSAInit(&BsaHandle, security_tokenPtr, &BsaObjectOwner, envx);
.
.
BSABeginTxn(BsaHandle);

/ * Delete the object from NetBackup. */

status = BSADeleteObject(BsaHandle, object_desc->copyId);
```

```
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSADeleteObject() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Delete");
    BSAEndTxn(BsaHandle, BSA_Vote_ABORT);
    BSATerminate(BsaHandle);
    exit(status);
}

/ * End the delete transaction, commit will delete the object */

status = BSAEndTxn(BsaHandle, BSA_Vote_COMMIT);
if (status != BSA_RC_SUCCESS) {
    Size = 512;
    NBBSAGetErrorString(status, &Size, ErrorString);
    sprintf(msg, "ERROR: BSAEndTxn() failed with error: %s", ErrorString);
    NBBSALogMsg(BsaHandle, MSERROR, msg, "Delete");
    BSATerminate(BsaHandle);
    exit(status);
}
```

Logging and NetBackup

NetBackup has a log directory that contains the debug logs for the various processes that make up the NetBackup server and/or client. There is a configurable verbose level that controls how much information is logged to these debug logs. This verbose level is a value from 0 to 5, with 0 indicating minimal logging and 5 being debug. These logs are used by NetBackup support to help solve customer problems. The log directory is located at `/usr/openv/netbackup/logs` on UNIX systems and *`install directory`*`/Veritas/NetBackup/logs` on Windows. Within this directory are directories for the different processes such as bprd, bpbrm, etc. One log file gets created for each day, and NetBackup automatically cleans up old files from this directory. The NetBackup XBSA Interface by default logs to the directory `exten_client`.

With NetBackup , some of the NetBackup services use Unified Logging (VxUL). Those include scheduler components: `nbjm`, `nbpem` and `nbrb`. For more details on VxUL please refer to the *NetBackup Troubleshooting Guide*. The NetBackup XBSA interface does not currently use VxUL.

The NetBackup XBSA Interface allows the XBSA Application to log in a manner consistent with other NetBackup processes. By using the NBBSALogMsg() function, the XBSA Application will log messages to the same file as the NetBackup XBSA

Interface. This may cause some confusion for the developer at first, especially at high debug levels, but allows the application to see what is causing errors and could help NetBackup support see what the XBSA Application is doing. The log messages contain a timestamp along with the process id, which is useful when there are multiple processes going at once.

The log message also contains a debug level. The different error levels used by NetBackup are defined in nbbsa.h. One of these values should be used in the msgType parameter of NBBSALogMsg(). While there are no hard definitions of when to use each of these values, using these values may help if NetBackup support or engineering is ever involved in looking at a debug log.

```
#define MSINFO          4
#define MSWARNING       8
#define MSERROR        16
#define MSCRITICAL     32
```

The XBSA Application is not required to log information to the NetBackup logs. If the XBSA Application is the backup portion of another application or database, it may make more sense to log information to a place consistent with the rest of the application.

Client in a cluster

Running an XBSA Application in a clustered environment is a valid mode of operation. The key thing about running in a cluster is to ensure that the client name used when an object is created is the same as the client name used when the object is being queried or retrieved. To ensure that the same client name is used, the XBSA Application should use the virtual name of the clients in the cluster. The best way to do this would be to use the NBBSA_CLIENT_HOST variable and set it to the virtual name for both creating and retrieving an object. The virtual name needs to be the client name that is configured in the NetBackup policy. Another option is to configure the virtual name as the default NetBackup client name. Configuring this way will then cause other NetBackup jobs, such as the file system backups, to use this virtual name also, which may not be desired. If neither of these options is used by the XBSA Application, the XBSA Interface will use the default client name, which will be the physical address of the client. What will happen then is that the objects will be created successfully, but if the query or retrieval is done from a different node in the cluster, the object will not be found.

Performance considerations

For the most part, the performance of the NetBackup XBSA Interface in conjunction with the XBSA Application is dependent on how NetBackup is configured and how

fast the XBSA Application can send or receive data. It is important that the developers of an XBSA Application have some understanding of NetBackup to get the most out of it. But much of that is determined by any individual implementation. But there are areas that the application can make a difference in performance.

Here are some hints to help you get the most out of the NetBackup XBSA Interface.

- Use copyId if you know it. If the XBSA Application has the ability to know or keep the copyId for further reference, this is the most efficient method of getting the object for restore.
- Specify dates when doing a query. If start and end dates are not specified when doing a query, the NetBackup XBSA Interface may need to search through the entire NetBackup catalog to find the object. Specifying dates allows it to narrow its search.
- Limit use of wildcards when doing a query. Wildcards, including the "ANY" value of the enumeration types, cause more overhead searching and can also cause large portions of the NetBackup catalog to be searched. This is especially true of the pathName. Wildcards can be very useful, but from a performance perspective they are harmful.
- Use OBJECT_STATUS_MOST_RECENT. If the XBSA application knows that a pathName is unique, or that it is searching for only the most recent copy of that object, set the objectStatus of the query descriptor to OBJECT_STATUS_MOST_RECENT. This will let NetBackup stop searching once one copy has been found.
- Unless the images are very large, create multiple objects per transaction rather than one object per transaction when there are multiple objects being created. Every transaction creates a NetBackup job that must be scheduled, open sockets, mount tapes, etc. For large objects, this overhead is dwarfed by the time it takes to backup the data. On the other hand if there are many small objects, this overhead becomes significant. Of course, creating multiple objects within one transaction limits the ability of the NetBackup XBSA Interface to delete an object.

Chapter 4

How to build an XBSA application

This chapter includes the following topics:

Getting help

Included in the NetBackup DataStore SDK are XBSA sample files that can be used as a basis for creating an application. Included are both source files and Makefiles. See the chapter What the sample files do for information on building and using the sample programs. The Makefiles included in the sample directory can be used as a basis for setting up an environment for creating an XBSA application.

Flags and defines

There are no specific flags or defines that need to be used in order to compile using the NetBackup XBSA Interface. You should be able to use any values to make your application compile efficiently.

How to build in debug mode

There is no compile level debug mode built into the XBSA libraries or header files. The NetBackup Verbose level controls debug messages.

How to debug the application

Debugging an XBSA application is best done through the log files generated by NetBackup. This assumes that the XBSA application itself compiles and does not have any known runtime errors.

See Logging and NetBackup for more information on this topic. You should also see the 'Logging' sections in the *NetBackup System Administrator's Guide for UNIX, Volume I*, or *NetBackup System Administrator's Guide for Windows, Volume I*,. The NetBackup Verbose level controls the amount of debug messages that are sent to the logs.

One of the advantages of debugging in this way is that it ties in with the NetBackup logging that is going on for the other NetBackup processes. In many cases, it could be a configuration issue that is causing a failure rather than a problem in the NetBackup XBSA interface or the XBSA application.

Static libraries

The example Makefiles have example entries for using static libraries for your XBSA application. These entries include the path to the static archive library, `libxbsa.a`, along with the system libraries that are also required to be included. For the platforms on which we support 32-bit binaries, a `libxbsa32.a` can be used to link to a 32-bit XBSA application.

For the UNIX platforms, (from `Makefile.unix`):

```
#LIBS    = $(XBSA_SDK_DIR)/lib/HP-UX-IA64/HP-UX11.31/libxbsa.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP-UX-IA64/HP-UX11.31/libxbsa32.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.11/libxbsa.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.11/libxbsa64.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.23/libxbsa.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.23/libxbsa64.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.31/libxbsa.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.31/libxbsa64.a
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/Debian2.6.18/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/Debian2.6.18/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMpSeriesRedHat2.6/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMpSeriesRedHat2.6/libxbsa32.a -lc -ldl
```

```
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMpSeriesSuSE2.6/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMpSeriesSuSE2.6/libxbsa32.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMzSeriesRedHat2.6.18/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMzSeriesSuSE2.6.18/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/RedHat2.6.18/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/RedHat2.6.18/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/SuSE2.6.16/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/SuSE2.6.16/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/RS6000/AIX6/libxbsa.a -ldl -lc
#LIBS    = $(XBSA_SDK_DIR)/lib/RS6000/AIX6/libxbsa64.a -ldl -lc
#LIBS    = $(XBSA_SDK_DIR)/lib/Solaris/Solaris10/libxbsa.a -lintl -lsocket -lnsl
-ldl -ladm
#LIBS    = $(XBSA_SDK_DIR)/lib/Solaris/Solaris10/libxbsa64.a -lintl -lsocket -lnsl
-ldl -ladm
#LIBS    = $(XBSA_SDK_DIR)/lib/Solaris/Solaris_x86_10_64/libxbsa.a -lintl -lsocket
-lnsl -ldl -ladm
```

For the Windows platforms, use:

```
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x86\Windows\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x86\Windows2003\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x86\Windows2008\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows2003\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows2008\xbsa.lib
```

Dynamic libraries

The example Makefiles have example entries for using dynamic libraries for your XBSA application.

For the UNIX platforms, (from `Makefile.unix`), choose the 32-bit or 64-bit dynamic library:

```
# Use one of these LIBS to bind dynamically

LIBS    = -L/usr/openv/lib -lxbsa -lnbclientcST -lnbbasecST
#LIBS    = -L/usr/openv/lib -lxbsa64 -lnbclientcST -lnbbasecST
#LIBS    = -L/usr/openv/lib -lxbsa32 -lnbclientcST32 -lnbbasecST32
```

For the Windows platforms, (from `Makefile.nt`):

```
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x86\Windows\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x86\Windows2003\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x86\Windows2008\xbsa.lib
```

```
#LIBS     = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows\xbsa.lib
#LIBS     = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows2003\xbsa.lib
#LIBS     = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows2008\xbsa.lib
```

The dynamic shared object libraries will be installed with the NetBackup 7.6 client on any supported client platform. Similar to the static libraries, on platforms that NetBackup supports both 32-bit and 64-bit applications, there will be a `libxbsa.so (sl)` and a `libxbsa64.so (sl)`. On UNIX platforms, the libraries are installed to `/usr/openv/lib`. On Windows platforms, the libraries are installed on *`install_directory`*`\netbackup\bin`.

End-user configuration

Once an XBSA application has been created and installed on a NetBackup Client, a NetBackup Policy and schedule must be configured through the NetBackup GUI or command line. See the chapter About How to run a NetBackup XBSA application.

Chapter 5

How to run a NetBackup XBSA application

This chapter includes the following topics:

About How to run a NetBackup XBSA application

Once an XBSA Application has been created, it can be used in a NetBackup environment to store and retrieve data. To use an XBSA application, at least one "DataStore" policy with the appropriate schedules needs to be defined. A configuration can have a single policy that includes all clients or there can be many policies, some of which include only one client.

This manual only contains a brief description of configuring a DataStore policy. More information on creating policies and configuring NetBackup can be found in the *NetBackup System Administrator's Guide for UNIX, Volume I*, or *NetBackup System Administrator's Guide for Windows, Volume I*.

Creating a NetBackup policy

A NetBackup policy defines the backup criteria for a specific group of one or more clients. These criteria include:

- storage unit and media to use
- backup schedules
- script files to be executed on the clients
- clients to be backed up

Selecting a storage unit

Each policy sends the data to a defined storage unit. The storage units must have already been defined and one needs to be selected for the DataStore policy.

Adding new schedules

Each policy has its own set of schedules. These schedules control initiation of automatic backups and also specify when user operations can be initiated.

A XBSA application requires each policy to have at least an Application Backup schedule. To help satisfy this requirement, an Application Backup schedule named Default-Application-Backup is automatically created when you configure a new DataStore policy. The backup window for an Application Backup schedule must encompass the time period during which all NetBackup DataStore jobs, scheduled and unscheduled, will occur. This is necessary because the Application Backup schedule starts processes that are required for all XBSA application backups, including those started automatically.

If the user wants NetBackup to initiate the XBSA application, an Automatic Backup schedule will also be required. An Automatic Backup schedule specifies the dates and times when NetBackup will automatically start backups by running the XBSA scripts in the order that they appear in the Files list. If there is more than one client in the DataStore policy, the XBSA scripts are executed on each client.

Adding script files to the files list

Each policy has a Files list. When a DataStore policy is configured, the Files list is actually a list of script(s) that are to be executed when the backup is initiated. That script that will be executed as a user-directed backup. Within the script should be any commands that are required to prepare the application for a backup, including setting up an environment, halting processes, etc., along with calling the XBSA Application with whatever parameters are required to execute the backup operations.

Adding new clients

Each policy also has a list of NetBackup clients. This list should contain all clients on which the XBSA application is going to run.

Running a NetBackup XBSA application

Once configured, backups and restores can be run either from the XBSA application or through jobs scheduled through NetBackup. NetBackup can run backups and

restores indirectly through the XBSA Application by executing scripts that contain XBSA Application backup or restore commands.

Backups and restores initiated by NetBackup (via a script)

The XBSA Application can be initiated through NetBackup. This allows the XBSA Application to be treated like the rest of the system environment when creating and scheduling backup windows and other resource considerations. Backup and restore operations through NetBackup are done via scripts. When a DataStore policy is configured, the Files list is actually a script that is to be executed when the backup or restore is initiated. That script that will be executed as a user-directed backup. Within these scripts should be any commands that are required to prepare the application for a backup or restore, including setting up an environment, halting processes, etc., along with calling the XBSA Application with whatever parameter are required to execute the backup or restore operations.

What is not available is the ability to browse for backups. The Files list is a script to be executed, not a list of objects that can be restored. It is up to these scripts to determine what needs to be backed up or conversely what XBSA objects need to be restored. In this regard, the XBSA Application needs to be fairly intelligent or allow options that can be specified to give the script the ability to be intelligent.

Backups and restores from the command line

The NetBackup XBSA Application can also be initiated from the command line to run a backup or restore. Commands included in the backup or restore scripts can also be run directly from the command line. The XBSA application will connect to NetBackup through the XBSA interface and a NetBackup job will be started. For backups, a backup window must be open in the Application Backup schedule.

Chapter 6

Process flow and troubleshooting

This chapter includes the following topics:

About Process flow and troubleshooting

The XBSA interface is provided to insulate the XBSA Applications from knowing about the internals of NetBackup and the processes and calls that are required to do backups and restores. This is appropriate when the application is working correctly. In the event that a problem occurs, this chapter gives a brief description of the NetBackup processes that get instantiated for each backup and restore. If process fails, a log is produced that contains information pertaining to the failure. All logs should be examined, however, for the root cause of the failure.

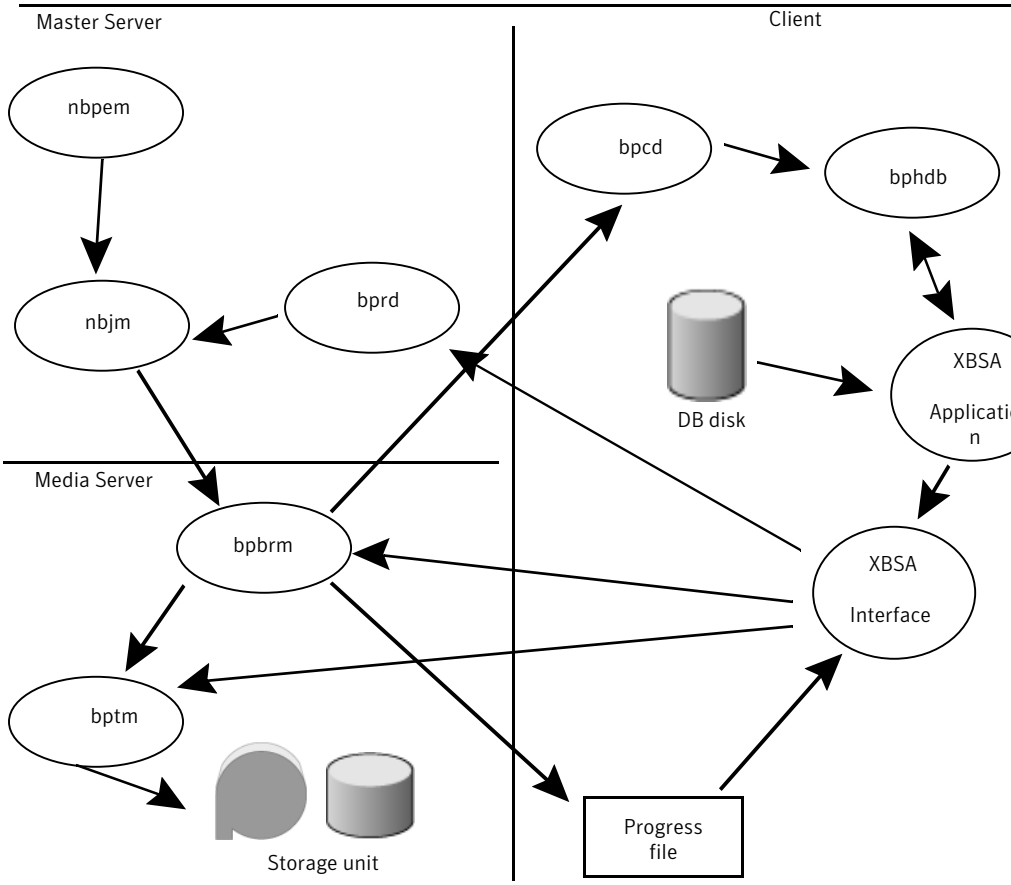
Backup

The backup stream diagram contains the processes involved in a backup being executed through a scheduled backup. bphdb is called to initiate the XBSA application, which then initiates one or more user-directed jobs.

If the backup is initiated from the XBSA application, it starts at that point. In the diagram below, the processes are divided among three logical machines, the Master Server, Media Server, and Client, but they could exist on only one or two machines.

Refer to the backup process diagrams in the "Functional Overview" appendix of the *NetBackup Troubleshooting Guide*.

Figure 6-1 Stream backup process flow diagram



Stream backup process flow description

This description provides a basic process flow. There are other processes involved that are not explicitly specified here. These processes include bpcd and some parent/child processes. Most connectivity from the server to the client initially goes through bpcd, such as initiating bphdb and writing to the progress log. And many of the processes, such as bpbrm, bptm, etc., initiate child processes. To keep the description simple, these processes are not included in the process steps.

Process steps

1. nbpem determines that a backup is scheduled to run and it initiates a backup job via nbjm.
2. nbjm starts bpbrm, which makes a request via bpcd.
3. bpbrm makes a request via the bpcd daemon to start bphdb on the client.
4. bphdb executes the backup script (which is contained in the Backup Selections list of the backup policy). Bphdb waits for an exit status from this script so that it can pass a status back to the server.
5. The backup script initiates the backup utility of the XBSA Application.

   (If it is not a scheduled backup operation, but is initiated on the client by the XBSA application, then the backup process starts here.)
6. The Application initiates the XBSA Interface by starting one or more sessions. Each session should be started in its own process. For simplicity, we will assume only one stream in this diagram. In reality, each stream follows each of these steps.
7. The backup is initiated with the first call to BSACreateObject(). This causes the XBSA Interface to make a bprd request to initiate a backup.
8. bprd submits a backup request to nbpem, which submits a job for nbjm. If this was a scheduled backup, there are now two backup jobs. nbjm initiates a bpbrm and a bpdbm process.
9. bpbrm initiates a bptm/bpdm process (bptm if tape storage unit, bpdm if disk storage unit). bptm initiates the process to mount media.
10. bpbrm writes progress information to the progress file on the client (via bpcd). This information includes sockets, status, backup attributes, etc.
11. XBSA reads the progress file to find the sockets and other information and connects to bpbrm on the name socket. It continues to read the progress file until it gets the message that it can continue the backup.
12. XBSA connects to bptm/bpdm through shared memory (if applicable) or on the data socket if the client and media server are separate machines.
13. XBSA sends the XBSA object entry to bpbrm, which sends it on to bpdbm to be catalogued.
14. At this point, BSACreateObject() returns to the XBSA Application. XBSA is ready to receive data.
15. The Application fills buffers and calls BSASendData() to have the XBSA Interface send these buffers to bptm/bpdm through the established connection.

16 bptm/bpdm writes this data to media or disk storage.

17 When the Application has sent all data, it indicates this with a BSAEndData() call. XBSA recognizes the object is complete.

18 The XBSA Application can then call BSACreateObject() again to create more objects. The subsequent CreateObject calls do not cause new jobs or connections, but continue with the existing processes.

19 When the XBSA Application has completed creating objects, a call to BSAEndTxn() causes XBSA to initiate the termination process. XBSA sends a client status to bpbrm, which terminates the server processes (bptm).

20 bpbrm writes the server status to the progress log and is read by XBSA. This allows the XBSA interface to confirm that the image has been successfully catalogued and all data written to media.

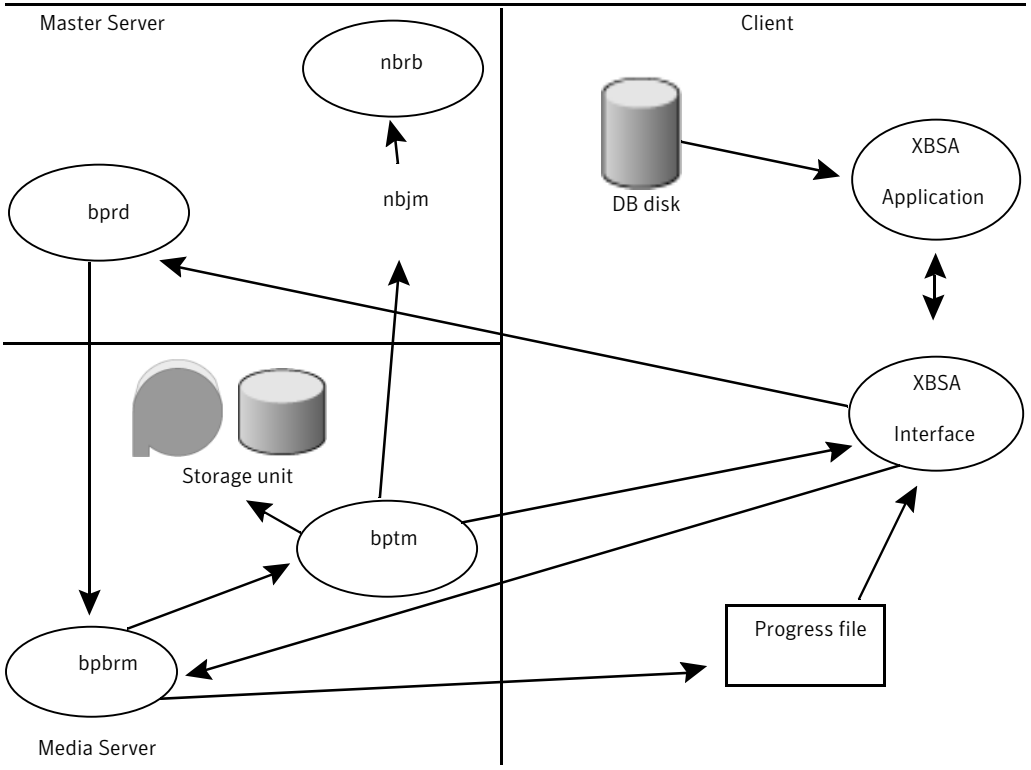
21 XBSA then passes this status back to the Application through the return status from BSAEndTxn(). This status is passed back to bphdb, which passes it back to the original scheduled job to complete the backup. This status displays in the Activity Monitor for the originally scheduled job.

Restore

The Restore diagram contains the processes involved in a restore that is executed from the XBSA Application. The restore process is very similar to the backup process except the flow of data is reversed. While it is possible to run a scheduled restore through bphdb, this is not a standard procedure; this diagram starts with the XBSA Application initiating the restore.

Refer to the restore process diagrams in the "Functional Overview" appendix of the *NetBackup Troubleshooting Guide*.

Figure 6-2 Stream restore process flow diagram



Stream restore process flow description

Similar to the backup diagram, the restore diagram is simple, without some of the parent/child processes. Note that the query process, which can be included in a restore operation, is not described in the following process. This process assumes the XBSA Application has the object descriptor to be passed to BSAGetObject.

Process steps

1. The XBSA Application initiates the XBSA Interface by starting one or more sessions. Each session is started in its own process. For simplicity, we will assume only one stream in this diagram. In reality, each stream will have each of these steps.
2. The restore is initiated with the first call to BSAGetObject(). This will cause the XBSA Interface to make a bprd request to initiate a restore.
3. bprd initiates a bpbrm process.

4. bpbrm initiates a bptm/bpdm process (bptm if tape storage unit, bpdm if disk storage unit). bptm gets the resources from nbjm/nbrb and initiates the process to mount media and start reading the data.
5. bpbrm writes progress information to the progress file on the client (via bpcd). This information includes sockets, status, restore attributes, etc.
6. XBSA reads the progress file to find the sockets and other information and connects to bpbrm on the name socket. It continues to read the progress file until it gets the message that it can continue the restore.
7. XBSA connects to bptm/bpdm via shared memory (if applicable) or on the data socket if the client and media server are separate machines.
8. At this point, BSAGetObject() returns to the XBSA Application. XBSA is ready to receive data.
9. The Application passes buffers to BSASendData() to have the XBSA Interface fill these buffers with data from bptm/bpdm via the established connection.
10. bptm/bpdm continues to read this data from media or disk storage and write it to to the buffers.
11. When the Application has received all data, it indicates this with a BSAEndData() call. XBSA verifies that all of the data from the object has been sent. XBSA sends a client status to bpbrm, which will then terminate the server processes (bptm).
12. bpbrm write the server status to the progress log and is read by XBSA. This allows the XBSA interface to confirm that the server has successfully read all the data and terminated.
13. The restore has been completed at this time. A call to BSAEndTxn() is required to close the transactions, but other than some internal cleanup, it does not provide any function for restores.

Chapter 7

How to use the sample files

This chapter includes the following topics:

What the sample files do

Included in the SDK are some simple sample programs and scripts. The sample programs can be used as examples of how to use the XBSA functions to create an XBSA application. The sample scripts are examples of how an XBSA application can be executed from a NetBackup schedule.

Sample programs

The SDK includes some simple sample programs that can be used as an example of the sequence of function calls that are required to create new objects, query the NetBackup database for existing objects, retrieve the objects, and delete objects. There is a separate program for each of these functions, although this is for the convenience of the samples and not necessarily a recommended way of building an XBSA application.

These programs as installed will not run. First, they need to be modified to set the correct hostname of the NetBackup server. Then they can be compiled and each can be individually run. Below is the description of the programs and what to expect from them if they have not been modified other than setting the hostname.

The following section of the sample programs needs to be modified. The entries ‘server_host’, ‘sample_policy’, and ‘sample_schedule’ need to be replaced with

actual values from your environment. Or these three entries can be eliminated so that the sample program uses default values from the NetBackup configuration.

```
/* Populate the XBSA environment variables for this session. */

strcpy(envx[0], "BSA_API_VERSION=1.1.0");
strcpy(envx[1], "BSA_SERVICE_HOST=server_host");
strcpy(envx[2], "NBBSA_POLICY=sample_policy");
strcpy(envx[3], "NBBSA_SCHEDULE=sample_schedule");
envx[4] = NULL;
```

Backup

This program will create one small object. The unique identifier, copyId, will be printed out along with the number of bytes backed up.

```
copyId: 1 - 1018898698
Bytes backed up: 154
```

Restore

This program will retrieve the last object created. The copyId will be printed out along with the text of the object data and the number of bytes that were retrieved.

```
Retrieving copyId: 1 - 1018898698
This is the sample data that is contained in the sample object that
is being backed up for the purposes of showing how data can be
backed up and restored.
Total bytes retrieved: 154
```

Query

This program will search for all objects created by the Backup program. The copyId of each of these objects will be printed out.

```
copyId: 1 - 1018898698
copyId: 1 - 1018898638
```

Delete

This program will delete the last object created. The copyId of the object being deleted will be printed out.

```
Deleting copyId: 1 - 1018898698
```

Sample scripts

Also included are some examples of scripts that can be used to initiate an XBSA Application as a scheduled NetBackup job. Again these are very simple scripts based on the sample programs. There are sample scripts for UNIX platforms (`*.sh`) and for Windows platforms (`*.cmd`).

In general use, the XBSA Application would have parameters or use system environment variables to communicate the parameters about the backup or restore operations. See the Running an XBSA application chapter for a better explanation of how these scripts work.

Description of sample files

This section includes a description of the sample files provided with the SDK. All sample files are located in `~sdk/DataStore/XBSA/samples`.

Table 7-1 Description of Sample Files

Filename	Description
Backup.c	This is an example of the functions needed to create an XBSA object.
Query.c	This is an example of the functions needed to search for an XBSA object.
Restore.c	This is an example of the functions needed to retrieve an XBSA object.
Delete.c	This is an example of the functions needed to delete an XBSA object.
Makefile.unix	This is an example Makefile which can be used to compile the sample programs on the UNIX platforms.
Makefile.nt	This is an example Makefile which can be used to compile the sample programs on Windows platforms.
backup_script.cmd	This is an example of the script needed to run an XBSA application from a NetBackup schedule on a Windows platform.
restore_script.cmd	This is an example of the script needed to run an XBSA application from a NetBackup schedule on a Windows platform.
backup_script.sh	This is an example of the script needed to run an XBSA application from a NetBackup schedule on a UNIX platform.

Table 7-1 Description of Sample Files *(continued)*

Filename	Description
restore_script.sh	This is an example of the script needed to run an XBSA application from a NetBackup schedule on a UNIX platform.

How to build the sample programs

Also included with the samples are a Makefile for UNIX platforms, Makefile.unix, and one for Windows, Makefile.nt. The Makefiles will compile the four sample programs using basic compiler options.

The UNIX Makefile needs to be modified to select which library to use. Library paths for all supported platforms are in the Makefile but commented out. The library for the required operating system needs to be chosen along with whether to use an archive library or a shared library.

The following lines are from `Makefile.unix`. One of the CFLAGS and one of the LIBS definitions need to be uncommented. The default is to compile 32 bit using the dynamic shared libraries.

The CFLAGS definitions are compile options. Select a CFLAGS definition for the system that is being compiled on. Note that this is a very minimal set of options and you may want to add other compile options based on your environment.

```
# Uncomment the CFLAGS for the environment that is being compiled
# General 32 bit
CFLAGS  =

# Solaris sparc 32 bit
#CFLAGS = -xarch=generic

# Solaris sparc 64 bit
#CFLAGS = -xarch=v9

# Solaris Opteron 64 bit
#CFLAGS = -xtarget=opteron -xarch=generic64

# HP PARISC 32 bit
#CFLAGS = +DA1.1 +DS2.0

# HP 64 bit
#CFLAGS = +DA2.0W +DS2.0
```

```
# HP IA64 bit
#CFLAGS = -Ae +DSitanium2 +DD64

# AIX 32 bit
#CFLAGS = -q32

# AIX 64 bit
#CFLAGS = -q64

# Linux x86 32 bit
#CFLAGS = -m32

# Linux x86 64 bit
#CFLAGS = -m64

# Linux on Power PC 64 bit
#CFLAGS = -mpowerpc64 -m64

# Linux on Power PC 32 bit
#CFLAGS = -mpowerpc64

DEFINES =
INCLUDES= -I$(XBSA_SDK_DIR)/include
```

The LIBS definitions define which XBSA library to link with. A shared object library is installed in `/usr/openv/lib` on all NetBackup clients and can be used for dynamic linking. An archive library for each platform is included in the SDK and can be used to statically link the application. Select a LIBS definition for the system that is being compiled. Be sure to use the 32-bit libraries if you are using 32-bit compile options.

```
# Use one of these LIBS to bind dynamically

#LIBS    = -L/usr/openv/lib -lxbsa -lnbclientcST -lnbbasecST
#LIBS    = -L/usr/openv/lib -lxbsa64 -lnbclientcST -lnbbasecST
#LIBS    = -L/usr/openv/lib -lxbsa32 -lnbclientcST32 -lnbbasecST32

# Or choose the correct LIBS for your system to bind statically

#LIBS    = $(XBSA_SDK_DIR)/lib/HP-UX-IA64/HP-UX11.31/libxbsa.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP-UX-IA64/HP-UX11.31/libxbsa32.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.11/libxbsa.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.11/libxbsa64.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.23/libxbsa.a
```

```
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.23/libxbsa64.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.31/libxbsa.a
#LIBS    = $(XBSA_SDK_DIR)/lib/HP9000-800/HP-UX11.31/libxbsa64.a
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/Debian2.6.18/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/Debian2.6.18/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMpSeriesRedHat2.6.18/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMpSeriesRedHat2.6.18/libxbsa32.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMpSeriesSuSE2.6.18/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMpSeriesSuSE2.6.18/libxbsa32.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMzSeriesRedHat2.6/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/IBMzSeriesSuSE2.6/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/RedHat2.6/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/RedHat2.6/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/SuSE2.6.16/libxbsa.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/Linux/SuSE2.6.16/libxbsa64.a -lc -ldl
#LIBS    = $(XBSA_SDK_DIR)/lib/RS6000/AIX6/libxbsa.a -ldl -lc
#LIBS    = $(XBSA_SDK_DIR)/lib/RS6000/AIX6/libxbsa64.a -ldl -lc
#LIBS    = $(XBSA_SDK_DIR)/lib/Solaris/Solaris10/libxbsa.a -lintl -lsocket -lnsl
-ldl -ladm
#LIBS    = $(XBSA_SDK_DIR)/lib/Solaris/Solaris10/libxbsa64.a -lintl -lsocket -lnsl
-ldl -ladm
#LIBS    = $(XBSA_SDK_DIR)/lib/Solaris/Solaris_x86_10_64/libxbsa.a -lintl -lsocket
-lnsl -ldl -ladm
```

The Windows Makefile may need to be modified to select which Windows library to use. The Windows Makefile needs to be modified if SDK was installed into a directory other than the default `c:\Program Files`.

```
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x86\Windows\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x86\Windows2003\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x86\Windows2008\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows2003\xbsa.lib
#LIBS    = $(XBSA_SDK_DIR)\lib\Windows-x64\Windows2008\xbsa.lib
```

Chapter 8

Support and updates

This chapter includes the following topics:

About Support and updates

The NetBackup SDK for DataStore is sold and distributed under specific licensing agreements. These licensing agreements will define how the SDK is supported, who to contact for support, and how upgrades will be supported. The agreements should also define how an XBSA application is sold and supported in conjunction with NetBackup. Please review your licensing agreement for details on product support.

Index

P

Q

R

S

T

X