AFS-3 Programmer's Reference: *BOS Server* Interface

Edward R. Zayas

Transarc Corporation

Version 1.0 of 28 August 1991 11:58
©Copyright 1991 Transarc Corporation
All Rights Reserved
FS-00-D161

Contents

Chapter 1

Overview

1.1 Introduction

One of the important duties of an AFS system administrator is to insure that processes on file server machines are properly installed and kept running. The *BOS Server* was written as a tool for assisting administrators in these tasks. An instance of the *BOS Server* runs on each AFS server machine, and has the following specific areas of responsibility:

- **Definition of the set of processes that are to be run** on the machine on which a given *BOS Server* executes. This definition may be changed dynamically by system administrators. Programs may be marked as continuously or periodically runnable.
- **Automatic startup and restart of these specified processes** upon server bootup and program failure. The *BOS Server* also responds to administrator requests for stopping and starting one or more of these processes. In addition, the *BOS Server* is capable of restarting itself on command.
- **Collection of information** regarding the current status, command line parameters, execution history, and log files generated by the set of server programs.
- **Management of the security information** resident on the machine on which the *BOS Server* executes. Such information includes the list of administratively privileged people associated with the machine and the set of AFS *File Server* encryption keys used in the course of file service.
- **Management of the cell configuration information** for the server machine in question. This includes the name of the cell in which the server resides, along

with the list and locations of the servers within the cell providing AFS database services (e.g., volume location, authentication, protection).

- **Installation of server binaries** on the given machine. The *BOS Server* allows several "generations" of server software to be kept on its machine. Installation of new software for one or more server agents is handled by the *BOS Server*, as is "rolling back" to a previous version should it prove more stable than the currently-installed image.
- **Execution of commands on the server machine**. An administrator may execute arbitrary UNIX commands on a machine running the *BOS Server*.

Unlike many other AFS server processes, the *BOS Server* does *not* maintain a cell-wide, replicated database. It does, however, maintain several databases used exclusively on every machine on which it runs.

1.2 Scope

This paper describes the design and structure of the AFS-3 *BOS Server*. The scope of this work is to provide readers with a sufficiently detailed description of the *BOS Server* so that they may construct client applications that call the server's RPC interface routines.

1.3 Document Layout

The second chapter discusses various aspects of the *BOS Server*'s architecture. First, one of the basic concepts is examined, namely the **bnode**. Providing the complete description of a program or set of programs to be run on the given server machine, a bnode is the generic definitional unit for the *BOS Server*'s duties. After bnodes have been explained, the set of standard directories on which the *BOS Server* depends is considered. Also, the set of well-known files within these directories is explored. Their uses and internal formats are presented. After these sections, a discussion of *BOS Server* restart times follows. The *BOS Server* has special support for two commonly-used restart occasions, as described by this section. Finally, the organization and behavior of the *bosserver* program itself is presented.

The third and final chapter provides a detailed examination of the programmer-visible *BOS Server* constants and structures, along with a full specification of the API for the RPC-based *BOS Server* functionality.

1.4 Related Documents

This document is a member of a documentation suite providing programmer-level specifications for the operation of the various AFS servers and agents, and the interfaces they export, as well as the underlying RPC system they use to communicate. The full suite of related AFS specification documents is listed below:

- *AFS-3 Programmer's Reference: Architectural Overview*: This paper provides an architectual overview of the AFS distributed file system, describing the full set of servers and agents in a coherent way, illustrating their relationships to each other and examining their interactions.
- *AFS-3 Programmer's Reference: File Server/Cache Manager Interface*: This document describes the *File Server* and *Cache Manager* agents, which provide the backbone file managment services for AFS. The collection of *File Server*s for a cell supply centralized file storage for that site, and allow clients running the *Cache Manager* component to acces those files in a high-performance, secure fashion.
- *AFS-3 Programmer's Reference:Volume Server/Volume Location Server Interface*: This document describes the services through which "containers" of related user data are located and managed.
- *AFS-3 Programmer's Reference: Protection Server Interface*: This paper describes the server responsible for mapping printable user names to and from their internal AFS identifiers. The *Protection Server* also allows users to create, destroy, and manipulate "groups" of users, which are suitable for placement on ACLs.
- *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*: This document specifies the design and operation of the remote procedure call and lightweight process packages used by AFS.

In addition to these papers, the AFS 3.1 product is delivered with its own user, administrator, installation, and command reference documents.

Chapter 2

BOS Server **Architecture**

This chapter considers some of the architectual features of the AFS-3 *BOS Server*. First, the basic organizational and functional entity employed by the *BOS Server*, the **bnode**, is discussed. Next, the set of files with which the server interacts is examined. The notion of **restart times** is then explained in detail. Finally, the organization and components of the *bosserver* program itself, which implements the *BOS Server*, are presented.

2.1 Bnodes

2.1.1 Overview

The information required to manage each AFS-related program running on a *File Server* machine is encapsulated in a **bnode** object. These bnodes serve as the basic building blocks for *BOS Server* services. Bnodes have two forms of existence:

- **On-disk**: The *BosConfig* file (see Section 2.3.4 below) defines the set of bnodes for which the *BOS Server* running on that machine will be responsible, along with specifying other information such as values for the two restart times. This file provides permanent storage (i.e., between bootups) for the desired list of programs for that server platform.
- **In-memory**: The contents of the *BosConfig* file are parsed and internalized by the *BOS Server* when it starts execution. The basic data for a particular server program is placed into a `struct bnode` structure.

The initial contents of the *BosConfig* file are typically set up during system installation. The *BOS Server* can be directed, via its RPC interface, to alter existing bnode entries in the *BosConfig* file, add new ones, and delete old ones. Typically, this file is never edited directly.

2.1.2 Bnode Classes

The descriptions of the members of the AFS server suite fall into three broad classes of programs:

- **Simple programs**: This server class is populated by programs that simply need to be kept running, and do not depend on other programs for correctness or effectiveness. Examples of AFS servers falling into this category are the *Volume Location Server*, *Authentication Server*, and *Protection Server*. Since its members exhibit such straightforward behavior, this class of programs is referred to as the **simple** class.
- **Interrelated programs**: The *File Server* program depends on two other programs, and requires that they be executed at the appropriate times and in the appropriate sequence, for correct operation. The first of these programs is the *Volume Server*, which must be run concurrently with the *File Server*. The second is the *salvager*, which repairs the AFS volume metadata on the server partitions should the metadata become damaged. The *salvager* **must not** be run at the same time as the *File Server*. In honor of the *File Server* trio that inspired it, the class of programs consisting of groups of interrelated processes is named the **fs** class.
- **Periodic programs**: Some AFS servers, such as the *BackupServer*, only need to run every so often, but on a regular and well-defined basis. The name for this class is taken from the UNIX tool that is typically used to define such regular executions, namely the **cron** class.

The recognition and definition of these three server classes is exploited by the *BOS Server*. Since all of the programs in a given class share certain common characteristics, they may all utilize the same basic data structures to record and manage their special requirements. Thus, it is not necessary to reimplement all the operations required to satisfy the capabilities promised by the *BOS Server* RPC interface for each and every program the *BOS Server* manages. Implementing one set of operations for each server class is sufficient to handle any and all server binaries to be run on the platform.

2.1.3 Per-Class Bnode Operations

As mentioned above, only one set of basic routines must be implemented for each AFS server class. Much like Sun's *VFS/vnode* interface [8], providing a common set of routines for interacting with a given file system, regardless of its underlying implementation and semantics, the *BOS Server* defines a common vector of operations for a class of programs to be run under the *BOS Server*'s tutelage. In fact, it was this standardized file system interface that inspired the "bnode" name.

The *BOS Server* manipulates the process or processes that are described by each bnode by invoking the proper functions in the appropriate order from the operation vector for that server class. Thus, the *BOS Server* itself operates in a class-independent fashion. This allows each class to take care of the special circumstances associated with it, yet to have the *BOS Server* itself be totally unaware of what special actions (if any) are needed for the class. This abstraction also allows more server classes to be implemented without any significant change to the *BOS Server* code itself.

There are ten entries in this standardized class function array. The purpose and usage of each individual class function is described in detail in Section 3.3.5. Much like the VFS system, a collection of macros is also provided in order to simplify the invocation of these functions. These macros are described in Section 3.5. The ten function slots are named here for convenience:

- *create()*
- *timeout()*
- *getstat()*
- *setstat()*
- *delete()*
- *procexit()*
- *getstring()*
- *getparm()*
- *restartp()*
- *hascore()*

2.2 *BOS Server* Directories

The *BOS Server* expects the existence of the following directories on the local disk of the platform on which it runs. These directories define where the system binaries, log files, *ubik* databases, and other files lie.

- */usr/afs/bin*: This directory houses the full set of AFS server binaries. Such executables as *bosserver*, *fileserver*, *vlserver*, *volserver*, *kaserver*, and *ptserver* reside here.
- */usr/afs/db*: This directory serves as the well-known location on the server's local disk for the *ubik* database replicas for this machine. Specifically, the *Authentication Server*, *Protection Server*, and the *Volume Location Server* maintain their local database images here.
- */usr/afs/etc*: This directory hosts the files containing the security, cell, and authorized system administrator list for the given machine. Specifically, the *CellServDB*, *KeyFile*, *License*, *ThisCell*, and *UserList* files are stored here.
- */usr/afs/local*: This directory houses the *BosConfig* file, which supplies the *BOS Server* with the permanent set of bnodes to support. Also contained in this directory are files used exclusively by the *salvager*.
- */usr/afs/logs*: All of the AFS server programs that maintain log files deposit them in this directory.

2.3 *BOS Server* Files

Several files, some mentioned above, are maintained on the server's local disk and manipulated by the *BOS Server*. This section examines many of these files, and describes their formats.

2.3.1 */usr/afs/etc/UserList*

This file contains the names of individuals who are allowed to issue "restricted" *BOS Server* commands (e.g., creating & deleting bnodes, setting cell information, etc.) on the given hardware platform. The format is straightforward, with one administrator name per line. If a cell grants `joe` and `schmoe` these rights on a machine, that particular *UserList* will have the following two lines:

```
joe
schmoe
```

2.3.2 */usr/afs/etc/CellServDB*

This file identifies the name of the cell to which the given server machine belongs, along with the set of machines on which its *ubik* database servers are running. Unlike the *CellServDB* found in */usr/vice/etc* on AFS client machines, this file contains *only* the entry for the home cell. It shares the formatting rules with the */usr/vice/etc* version of *CellServDB*. The contents of the *CellServDB* file used by the *BOS Server* on host `grand.central.org` are:

```
>grand.central.org          #DARPA clearinghouse cell
192.54.226.100              #grand.central.org
192.54.226.101              #penn.central.org
```

2.3.3 */usr/afs/etc/ThisCell*

The *BOS Server* obtains its notion of cell membership from the *ThisCell* file in the specified directory. As with the version of *ThisCell* found in */usr/vice/etc* on AFS client machines, this file simply contains the character string identifying the home cell name. For any server machine in the `grand.central.org` cell, this file contains the following:

```
grand.central.org
```

2.3.4 */usr/afs/local/BosConfig*

The *BosConfig* file is the on-disk representation of the collection of bnodes this particular *BOS Server* manages. The *BOS Server* reads and writes to this file in the normal course of its affairs. The *BOS Server* itself, in fact, should be the only agent that modifies this file. Any changes to *BosConfig* should be carried out by issuing the proper RPCs to the *BOS Server* running on the desired machine.

The following is the text of the *BosConfig* file on `grand.central.org`. A discussion of the contents follows immediately afterwards.

```
restarttime 11 0 4 0 0
checkbintime 3 0 5 0 0
bnode simple kaserver 1
parm /usr/afs/bin/kaserver
end
bnode simple ptserver 1
parm /usr/afs/bin/ptserver
end
bnode simple vlserver 1
parm /usr/afs/bin/vlserver
end
bnode fs fs 1
parm /usr/afs/bin/fileserver
parm /usr/afs/bin/volserver
parm /usr/afs/bin/salvager
end
bnode simple runntp 1
parm /usr/afs/bin/runntp -localclock transarc.com
end
bnode simple upserver 1
parm /usr/afs/bin/upserver
end
bnode simple budb_server 1
parm /usr/afs/bin/budb_server
end
bnode cron backup 1
parm /usr/afs/backup/clones/lib/backup.csh daily
parm 05:00
end
```

The first two lines of this file set the system and new-binary restart times (see Section 2.4, below). They are optional, with the default system restart time being every Sunday at 4:00am and the new-binary restart time being 5:00am daily. Following the reserved words `restarttime` and `checkbintime` are five integers, providing the mask, day, hour, minute, and second values (in decimal) for the restart time, as diagrammed below:

```
restarttime  <mask> <day> <hour> <minute> <second>
checkbintime <mask> <day> <hour> <minute> <second>
```

The range of acceptable values for these fields is presented in Section 3.3.1. In this example, the `restart` line specifies a (decimal) mask value of 11, selecting the `KTIME_HOUR`, `KTIME_MIN`, and `KTIME_DAY` bits. This indicates that the hour, minute, and day values are the ones to be used when matching times. Thus, this line requests that system restarts occur on day 0 (Sunday), hour 4 (4:00am), and minute 0 within that hour.

The sets of lines that follow define the individual bnodes for the particular machine. The first line of the bnode definition set must begin with the reserved word `bnode`, followed by the type name, the instance name, and the initial bnode goal:

```
bnode <type_name> <instance_name> <goal_val>
```

The <type_name> and <instance_name> fields are both character strings, and the <goal_val> field is an integer. Acceptable values for the <type_name> are `simple`, `fs`, and `cron`. Acceptable values for <goal_val> are defined in Section 3.2.3, and are normally restricted to the integer values representing BSTAT_NORMAL and BSTAT_SHUTDOWN. Thus, in the bnode line defining the *Authentication Server*, it is declared to be of type `simple`, have instance name `kaserver`, and have 1 (BSTAT_NORMAL) as a goal (e.g., it should be brought up and kept running).

Following the `bnode` line in the *BosConfig* file may be one or more `parm` lines. These entries represent the command line parameters that will be used to invoke the proper related program or programs. The entire text of the line after the `parm` reserved word up to the terminating newline is stored as the command line string.

```
parm <command_line_text>
```

After the `parm` lines, if any, the reserved word `end` must appear alone on a line, marking the end of an individual bnode definition.

2.3.5 */usr/afs/local/NoAuth*

The appearance of this file is used to mark whether the *BOS Server* is to insist on properly authenticated connections for its restricted operations or whether it will allow any caller to exercise these commands. Not only is the *BOS Server* affected by the presence of this file, but so are all of the bnodes objects the *BOS Server* starts up. If */usr/afs/local/NoAuth* is present, the *BOS Server* will start all of its bnodes with the `-noauth` flag.

Completely unauthenticated AFS operation will result if this file is present when the *BOS Server* starts execution. The file itself is typically empty. If any data is put into the *NoAuth* file, it will be ignored by the system.

2.3.6 */usr/afs/etc/KeyFile*

This file stores the set of AFS encryption keys used for file service operations. The file follows the format defined by `struct afsconf_key` and `struct afsconf_keys` in include file *afs/keys.h*. For the reader's convenience, these structures are detailed below:

```
#define AFSCONF_MAXKEYS 8

struct afsconf_key {
    long kvno;
    char key[8];
};

struct afsconf_keys {
    long nkeys;
    struct afsconf_key key[AFSCONF_MAXKEYS];
};
```

The first longword of the file reveals the number of keys that may be found there, with a maximum of AFSCONF_MAXKEYS (8). The keys themselves follow, each preceded by its key version number.

All information in this file is stored in network byte order. Each *BOS Server* converts the data to the appropriate host byte order befor storing and manipulating it.

2.4 Restart Times

It is possible to manually start or restart any server defined within the set of *BOS Server* bnodes from any AFS client machine, simply by making the appropriate call to the RPC interface while authenticated as a valid administrator (i.e., a principal listed in the *UserList* file on the given machine). However, two restart situations merit the implementation of special functionality within the *BOS Server*. There are two common occasions, occuring on a regular basis, where the entire system or individual server programs should be brought down and restarted:

- **Complete system restart**: To guard against the reliability and performance problems caused by any core leaks in long-running programs, the entire AFS system should be occasionally shut down and restarted periodically. This action "clears out" the memory system, and may result in smaller memory images for these servers, as internal data structures are reinitialized back to their starting sizes. It also allows AFS partitions to be regularly examined, and salvaged if necessary.

  Another reason for performing a complete system restart is to commence execution of a new release of the *BOS Server* itself. The new-binary restarts described below do not restart the *BOS Server* if a new version of its software has been installed on the machine.

- **New-binary restarts**: New server software may be installed at any time with the assistance of the *BOS Server*. However, it is often not the case that such

> software installations occur as a result of the discovery of an error in the program or programs requiring immediate restart. On these occasions, restarting the given processes in prime time so that the new binaries may begin execution is counterproductive, causing system downtime and interfering with user productivity. The system administrator may wish to set an off-peak time when the server binaries are automatically compared to the running program images, and restarts take place should the on-disk binary be more recent than the currently running program. These restarts would thus minimize the resulting service disruption.

Automatically performing these restart functions could be accomplished by creating `cron`-type bnodes that were defined to execute at the desired times. However, rather than force the system administrator to create and supervise such bnodes, the *BOS Server* supports

the notion of an internal LWP thread with the same effect (see Section 2.5.2). As part of the *BosConfig* file defined above, the administrator may simply specify the values for the complete system restart and new-binary restart times, and a dedicated *BOS Server* thread will manage the restarts.

Unless otherwise instructed, the *BOS Server* selects default times for the two above restart times. A complete system restart is carried out every Sunday at 4:00am by default, and a new-binary restart is executed for each updated binary at 5:00am every day.

2.5 The *bosserver* Process

2.5.1 Introduction

The user-space *bosserver* process is in charge of managing the AFS server processes and software images, the local security and cell database files, and allowing administrators to execute arbitrary programs on the server machine on which it runs. It also implements the RPC interface defined in the *bosint.xg Rxgen* definition file.

2.5.2 Threading

As with all the other AFS server agents, the *BOS Server* is a multithreaded program. It is configured so that a minimum of two lightweight threads are guaranteed to be allocated to handle incoming RPC calls to the *BOS Server*, and a maximum of four threads are commissioned for this task.

In addition to these threads assigned to RPC duties, there is one other thread employed by the *BOS Server*, the *BozoDaemon()*. This thread is responsible for keeping track of the two major restart events, namely the system restart and the new binary restart (see Section 2.4). Every 60 seconds, this thread is awakened, at which time it checks to see if either deadline has occurred. If the complete system restart is then due, it invokes internal *BOS Server* routines to shut down the entire suite of AFS agents on that machine and then reexecute the *BOS Server* binary, which results in the restart of all of the server processes. If the new-binary time has arrived, the *BOS Server* shuts down the bnodes for which binaries newer than those running are available, and then invokes the new software.

In general, the following procedure is used when stopping and restarting processes. First, the *restart()* operation defined for each bnode's class is invoked via the *BOP_RESTART()* macro. This allows each server to take any special steps required before cycling its service. After that function completes, the standard mechanisms are used to shut down each bnode's process, wait until it has truly stopped its execution, and then start it back up again.

2.5.3 Initialization Algorithm

This section describes the procedure followed by the *BOS Server* from the time when it is invoked to the time it has properly initialized the server machine upon which it is executing.

The first check performed by the *BOS Server* is whether or not it is running as `root`. It needs to manipulate local UNIX files and directories in which only root has been given access, so it immediately exits with an error message if this is not the case. The *BOS Server*'s UNIX working directory is then set to be */usr/afs/logs* in order to more easily service incoming RPC requests to fetch the contents of the various server log files at this location. Also, changing the working directory in this fashion results in any core images dumped by the *BOS Server*'s wards will be left in */usr/afs/logs*.

The command line is then inspected, and the *BOS Server* determines whether it will insist on authenticated RPC connections for secure administrative operations by setting up the */usr/afs/local/NoAuth* file appropriately (see Section 2.3.5). It initializes the underlying bnode package and installs the three known bnode types (`simple`, `fs`, and `cron`).

After the bnode package is thus set up, the *BOS Server* ensures that the set of local directories on which it will depend are present; refer to Section 2.2 for more details on this matter. The license file in */usr/afs/etc/License* is then read to determine the number of AFS server machines the site is allowed to operate, and whether the cell is allowed to run the NFS/AFS Translator software. This file is typically obtained in the initial system installation, taken from the installation tape. The *BOS Server* will exit unless this file exists and is properly formatted.

In order to record its actions, any existing */usr/afs/logs/BosLog* file is moved to *BosLog.old*, and a new version is opened in append mode. The *BOS Server* immediately writes a log entry concerning the state of the above set of important directories.

At this point, the *BOS Server* reads the *BosConfig* file, which lists the set of bnodes for which it will be responsible. It starts up the processes associated with the given bnodes. Once accomplished, it invokes its internal system restart LWP thread (covered in Section 2.5.2 above).

Rx initialization begins at this point, setting up the RPC infrastructure to receive its packets on the `AFSCONF_NANNYPORT`, UDP port 7007. The local cell database is then read and internalized, followed by acquisition of the AFS encryption keys.

After all of these steps have been carried out, the *BOS Server* has gleaned all of the necessary information from its environemnt and has also started up its wards. The final initialization action required is to start all of its listener LWP threads, which are devoted to executing incoming requests for the *BOS Server* RPC interface.

2.5.4 Command Line Switches

The *BOS Server* recognizes exactly one command line argument: `-noauth`. By default, the *BOS Server* attempts to use authenticated RPC connections (unless the */usr/afs/local/NoAuth* file is present; see Section 2.3.5). The appearance of the `-noauth` command line flag signals that this server incarnation is to use unauthenticated connections for even those operations that are normally restricted to system administrators. This switch is essential during the initial AFS system installation, where the procedures followed to bootstrap AFS onto a new machine require the *BOS Server* to run before some system databases have been created.

Chapter 3

BOS Server **Interface**

3.1 Introduction

This chapter documents the API for the *BOS Server* facility, as defined by the *bosint.xg Rxgen* interface file and the *bnode.h* include file. Descriptions of all the constants, structures, macros, and interface functions available to the application programmer appear in this chapter.

3.2 Constants

This section covers the basic constant definitions of interest to the *BOS Server* application programmer. These definitions appear in the *bosint.h* file, automatically generated from the *bosint.xg Rxgen* interface file. Another file is exported to the programmer, namely *bnode.h*.

Each subsection is devoted to describing constants falling into each of the following categories:

- Status bits
- Bnode activity bits
- Bnode states
- Pruning server binaries

- Flag bits for `struct bnode_proc`

One constant of general utility is `BOZO_BSSIZE`, which defines the length in characters of *BOS Server* character string buffers, including the trailing null. It is defined to be 256 characters.

3.2.1 Status Bits

The following bit values are used in the `flags` field of `struct bozo_status`, as defined in Section 3.3.4. They record whether or not the associated bnode process currently has a stored core file, whether the bnode execution was stopped because of an excessive number of errors, and whether the mode bits on server binary directories are incorrect.

Name	*Value*	*Description*
`BOZO_HASCORE`	1	Does this bnode have a stored core file?
`BOZO_ERRORSTOP`	2	Was this bnode execution shut down because of an excessive number of errors (more than 10 in a 10-second period)?
`BOZO_BADDIRACCESS`	3	Are the mode bits on the */usr/afs* directory and its descendants (*etc*, *bin*, *logs*, *backup*, *db*, *local*, *etc/KeyFile*, *etc/UserList*) correctly set?

3.2.2 Bnode Activity Bits

This section describes the legal values for the bit positions within the `flags` field of `struct bnode`, as defined in Section 3.3.8. They specify conditions related to the basic activity of the bnode and of the entities relying on it.

Name	*Value*	*Description*
`BNODE_NEEDTIMEOUT`	0x01	This bnode is utilizing the timeout mechanism for invoking actions on its behalf.
`BNODE_ACTIVE`	0x02	The given bnode is in active service.
`BNODE_WAIT`	0x04	Someone is waiting for a status change in this bnode
`BNODE_DELETE`	0x08	This bnode should be deleted at the earliest convenience.
`BNODE_ERRORSTOP`	0x10	This bnode decommissioned because of an excessive number of errors in its associated UNIX processes.

3.2.3 Bnode States

The constants defined in this section are used as values within the `goal` and `fileGoal` fields within a `struct bnode`. They specify either the current state of the associated bnode, or the anticipated state. In particular, the `fileGoal` field, which is the value stored on disk for the bnode, always represents the desired state of the bnode, whether or not it properly reflects the current state. For this reason, only `BSTAT_SHUTDOWN` and `BSTAT_NORMAL` may be used within the `fileGoal` field. The `goal` field may take on any of these values, and accurately reflects the current status of the bnode.

Name	*Value*	*Description*
`BSTAT_SHUTDOWN`	0	The bnode's execution has been (should be) terminated.
`BSTAT_NORMAL`	1	The bnode is (should be) executing normally.
`BSTAT_SHUTTINGDOWN`	2	The bnode is currently being shutdown; execution has not yet ceased.
`BSTAT_STARTINGUP`	3	The bnode execution is currently being commenced; execution has not yet begun.

3.2.4 Pruning Server Binaries

The *BOZO_Prune()* interface function, fully defined in Section 3.6.6.4, allows a properly-authenticated caller to remove ("prune") old copies of server binaries and core files managed by the *BOS Server*. This section identifies the legal values for the `flags` argument to the above function, specifying exactly what is to be pruned.

Name	*Value*	*Description*
`BOZO_PRUNEOLD`	1	Prune all server binaries with the **.OLD* extension.
`BOZO_PRUNEBAK`	2	Prune all server binaries with the **.BAK* extension.
`BOZO_PRUNECORE`	3	Prune core files.

3.2.5 Flag Bits for `struct bnode_proc`

This section specifies the acceptable bit values for the `flags` field in the `struct bnode_proc` structure, as defined in Section 3.3.9. Basically, they are used to record whether or not the UNIX binary associated with the bnode has ever been run, and if so whether it has ever exited.

Name	*Value*	*Description*
`BPROC_STARTED`	1	Has the associated UNIX process ever been started up?
`BPROC_EXITED`	2	Has the associated UNIX process ever exited?

3.3 Structures

This section describes the major exported *BOS Server* data structures of interest to application programmers.

3.3.1 `struct bozo_netKTime`

This structure is used to communicate time values to and from the *BOS Server*. In particular, the *BOZO_GetRestartTime()* and *BOZO_SetRestartTime()* interface functions, described in Sections 3.6.2.5 and 3.6.2.6 respectively, use parameters declared to be of this type.

Four of the fields in this structure specify the hour, minute, second, and day of the event in question. The first field in the layout serves as a mask, identifying which of the above four fields are to be considered when matching the specified time to a given reference time (most often the current time). The bit values that may be used for the mask field are defined in the *afs/ktime.h* include file. For convenience, their values are reproduced here, including some special cases at the end of the table.

Name	*Value*	*Description*
`KTIME_HOUR`	0x01	Hour should match.
`KTIME_MIN`	0x02	Minute should match.
`KTIME_SEC`	0x04	Second should match.
`KTIME_DAY`	0x08	Day should match.
`KTIME_TIME`	0x07	All times should match.
`KTIME_NEVER`	0x10	Special case: never matches.
`KTIME_NOW`	0x20	Special case: right now.

Fields

int mask - A field of bit values used to specify which of the following field are to be used in computing matches.

short hour - The hour, ranging in value from 0 to 23.

short min - The minute, ranging in value from 0 to 59.

short sec - The second, ranging in value from 0 to 59.

short day - Zero specifies Sunday, other days follow in order.

3.3.2 struct bozo_key

This structure defines the format of an AFS encryption key, as stored in the key file located at */usr/afs/etc/KeyFile* at the host on which the *BOS Server* runs. It is used in the argument list of the *BOZO_ListKeys()* and *BOZO_AddKeys()* interface functions, as described in Sections 3.6.4.4 and 3.6.4.5 respectively.

Fields

char data[8] - The array of 8 characters representing an encryption key.

3.3.3 struct bozo_keyInfo

This structure defines the information kept regarding a given AFS encryption key, as represented by a variable of type `struct bozo_key`, as described in Section 3.3.2 above. A parameter of this type is used by the *BOZO_ListKeys()* function (described in Section 3.6.4.4). It contains fields holding the associated key's modification time, a checksum on the key, and an unused longword field. Note that the `mod_sec` time field listed below is a standard UNIX time value.

Fields

long mod_sec - The time in seconds when the associated key was last modified.

long mod_usec - The number of microseconds elapsed since the second reported in the `mod_sec` field. This field is never set by the *BOS Server*, and should always contain a zero.

unsigned long keyCheckSum - The 32-bit cryptographic checksum of the associated key. A block of zeros is encrypted, and the first four bytes of the result are placed into this field.

long spare2 - This longword field is currently unused, and is reserved for future use.

3.3.4 struct bozo_status

This structure defines the layout of the information returned by the `status` parameter for the interface function *BOZO_GetInstanceInfo()*, as defined in Section 3.6.2.3. The enclosed fields include such information as the temporary and long-term goals for the process instance, an array of bit values recording status information, start and exit times, and associated error codes and signals.

Fields

- **long goal** - The short-term goal for a process instance. Settings for this field are `BSTAT_SHUTDOWN`, `BSTAT_NORMAL`, `BSTAT_SHUTTINGDOWN`, and `BSTAT_STARTINGUP`. These values are fully defined in Section 3.2.3.
- **long fileGoal** - The long-term goal for a process instance. Accepted settings are restricted to a subset of those used by the `goal` field above, as explained in Section 3.2.3.
- **long procStartTime** - The last time the given process instance was started.
- **long procStarts** - The number of process starts executed on the behalf of the given bnode.
- **long lastAnyExit** - The last time the process instance exited for any reason.
- **long lastErrorExit** - The last time a process exited unexpectedly.
- **long errorCode** - The last exit's return code.
- **long errorSignal** - The last signal terminating the process.
- **long flags** - `BOZO_HASCORE`, `BOZO_ERRORSTOP`, and `BOZO_BADDIRACCESS`. These constants are fully defined in Section 3.2.1.
- **long spare**[] - Eight longword spares, currently unassigned and reserved for future use.

3.3.5 struct bnode_ops

This struture defines the base set of operations that each *BOS Server* bnode type (`struct bnode_type`, see Section 3.3.6 below) must implement. They are called at the appropriate times within the *BOS Server* code via the *BOP_** macros (see Section 3.5 and the individual descriptions therein). They allow each bnode type to define its own behavior in response to its particular needs.

Fields

struct bnode *(*create)() - This function is called whenever a bnode of the given type is created. Typically, this function will create bnode structures peculiar to its own type and initialize the new records. Each type implementation may take a different number of parameters. Note: there is no *BOP_** macro defined for this particular function; it is always called directly.

int (*timeout)() - This function is called whenever a timeout action must be taken for this bnode type. It takes a single argument, namely a pointer to a type-specific bnode structure. The *BOP_TIMEOUT* macro is defined to simplify the construction of a call to this function.

int (*getstat)() - This function is called whenever a caller is attempting to get status information concerning a bnode of the given type. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a pointer to a longword in which the desired status value will be placed. The *BOP_GETSTAT* macro is defined to simplify the construction of a call to this function.

int (*setstat)() - This function is called whenever a caller is attempting to set the status information concerning a bnode of the given type. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a longword from which the new status value is obtained. The *BOP_SETSTAT* macro is defined to simplify the construction of a call to this function.

int (*delete)() - This function is called whenever a bnode of this type is being deleted. It is expected that the proper deallocation and cleanup steps will be performed here. It takes a single argument, a pointer to a type-specific bnode structure. The *BOP_DELETE* macro is defined to simplify the construction of a call to this function.

int (*procexit)() - This function is called whenever the UNIX process implementing the given bnode exits. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a pointer to the `struct bnode_proc` (defined in Section 3.3.9), describing that process in detail. The *BOP_PROCEXIT* macro is defined to simplify the construction of a call to this function.

int (*getstring)() - This function is called whenever the status string for the given bnode must be fetched. It takes three parameters. The first is a pointer to a type-specific bnode structure, the second is a pointer to a character buffer, and the third is a longword specifying the size, in bytes, of the above buffer. The *BOP_GETSTRING* macro is defined to simplify the construction of a call to this function.

int (*getparm)() - This function is called whenever a particular parameter string for the given bnode must be fetched. It takes four parameters. The first is a pointer to a type-specific bnode structure, the second is a longword identifying the index of the desired parameter string, the third is a pointer to a character buffer to receive the parameter string, and the fourth and final argument is a longword specifying the size, in bytes, of the above buffer. The *BOP_GETPARM* macro is defined to simplify the construction of a call to this function.

int (*restartp)() - This function is called whenever the UNIX process implementing the bnode of this type is being restarted. It is expected that the stored process command line will be parsed in preparation for the coming execution. It takes a single argument, a pointer to a type-specific bnode structure from which the command line can be located. The *BOP_RESTARTP* macro is defined to simplify the construction of a call to this function.

int (*hascore)() - This function is called whenever it must be determined if the attached process currently has a stored core file. It takes a single argument, a pointer to a type-specific bnode structure from which the name of the core file may be constructed. The *BOP_HASCORE* macro is defined to simplify the construction of a call to this function.

3.3.6 `struct bnode_type`

This structure encapsulates the defining characteristics for a given bnode type. Bnode types are placed on a singly-linked list within the *BOS Server*, and are identified by a null-terminated character string name. They also contain the function array defined in Section 3.3.5, that implements the behavior of that object type. There are three predefined bnode types known to the *BOS Server*. Their names are `simple`, `fs`, and `cron`. It is not currently possible to dynamically define and install new *BOS Server* types.

Fields

struct bnode_type *next - Pointer to the next bnode type definition structure in the list.

char *name - The null-terminated string name by which this bnode type is identified.

bnode_ops *ops - The function array that defines the behavior of this given bnode type.

3.3.7 struct bnode_token

This structure is used internally by the *BOS Server* when parsing the command lines with which it will start up process instances. This structure is made externally visible should more types of bnode types be implemented.

Fields

struct bnode_token *next - The next token structure queued to the list.

char *key - A pointer to the token, or parsed character string, associated with this entry.

3.3.8 struct bnode

This structure defines the essence of a *BOS Server* process instance. It contains such important information as the identifying string name, numbers concerning periodic execution on its behalf, the bnode's type, data on start and error behavior, a reference count used for garbage collection, and a set of flag bits.

Fields

char *name - The null-terminated character string providing the instance name associated with this bnode.

long nextTimeout - The next time this bnode should be awakened. At the specified time, the bnode's `flags` field will be examined to see if `BNODE_NEEDTIMEOUT` is set. If so, its *timeout()* operation will be invoked via the *BOP_TIMEOUT()* macro. This field will then be reset to the current time plus the value kept in the `period` field.

long period - This field specifies the time period between timeout calls. It is only used by processes that need to have periodic activity performed.

long rsTime - The time that the *BOS Server* started counting restarts for this process instance.

long rsCount - The count of the number of restarts since the time recorded in the `rsTime` field.

struct bnode_type *type - The type object defining this bnode's behavior.

struct bnode_ops *ops - This field is a pointer to the function array defining this bnode's basic behavior. Note that this is identical to the value of `type->ops`.

This pointer is duplicated here for convenience. All of the *BOP_** macros, discussed in Section 3.5, reference the bnode's operation array through this pointer.

long procStartTime - The last time this process instance was started (executed).

long procStarts - The number of starts (executions) for this process instance.

long lastAnyExit - The last time this process instance exited for any reason.

long lastErrorExit - The last time this process instance exited unexpectedly.

long errorCode - The last exit return code for this process instance.

long errorSignal - The last signal that terminated this process instance.

char *lastErrorName - The name of the last core file generated.

short refCount - A reference count maintained for this bnode.

short flags - This field contains a set of bit fields that identify additional status information for the given bnode. The meanings of the legal bit values, explained in Section 3.2.2, are: `BOZO_NEEDTIMEOUT`, `BOZO_ACTIVE`, `BOZO_WAIT`, `BOZO_DELETE`, and `BOZO_ERRORSTOP`.

char goal - The current goal for the process instance. It may take on any of the values defined in Section 3.2.3, namely `BSTAT_SHUTDOWN`, `BSTAT_NORMAL`, `BSTAT_SHUTTINGDOWN`, and `BSTAT_STARTINGUP`.

This goal may be changed at will by an authorized caller. Such changes affect the current status of the process instance. See the description of the *BOZO_SetStatus()* and *BOZO_SetTStatus()* interface functions, defined in Sections 3.6.3.1 and 3.6.3.2 respectively, for more details.

char fileGoal - This field is similar to `goal`, but represents the goal stored in the on-file *BOS Server* description of this process instance. As with the `goal` field, see functions the description of the *BOZO_SetStatus()* and *BOZO_SetTStatus()* interface functions defined in Sections 3.6.3.1 and 3.6.3.2 respectively for more details.

3.3.9 `struct bnode_proc`

This structure defines all of the information known about each UNIX process the *BOS Server* is currently managing. It contains a reference to the bnode defining the process, along with the command line to be used to start the process, the optional core file name, the UNIX pid, and such things as a flag field to keep additional state information. The *BOS Server* keeps these records on a global singly-linked list.

Fields

struct bnode_proc *next - A pointer to the *BOS Server*'s next process record.

struct bnode *bnode - A pointer to the bnode creating and defining this UNIX process.

char *comLine - The text of the command line used to start this process.

char *coreName - An optional core file component name for this process.

long pid - The UNIX pid, if successfully created.

long lastExit - This field keeps the last termination code for this process.

long lastSignal - The last signal used to kill this process.

long flags - A set of bits providing additional process state. These bits are fully defined in Section 3.2.5, and are: `BPROC_STARTED` and `BPROC_EXITED`.

3.4 Error Codes

This section covers the set of error codes exported by the *BOS Server*, displaying the printable phrases with which they are associated.

Name	*Value*	*Description*
`BZNOTACTIVE`	(39424L)	process not active.
`BZNOENT`	(39425L)	no such entity.
`BZBUSY`	(39426L)	can't do operation now.
`BZEXISTS`	(39427L)	entity already exists.
`BZNOCREATE`	(39428L)	failed to create entity.
`BZDOM`	(39429L)	index out of range.
`BZACCESS`	(39430L)	you are not authorized for this operation.
`BZSYNTAX`	(39431L)	syntax error in create parameter.
`BZIO`	(39432L)	I/O error.
`BZNET`	(39433L)	network problem.
`BZBADTYPE`	(39434L)	unrecognized bnode type.

3.5 Macros

The *BOS Server* defines a set of macros that are externally visible via the *bnode.h* file. They are used to facilitate the invocation of the members of the `struct bnode_ops` function array, which defines the basic operations for a given bnode type. Invocations appear throughout the *BOS Server* code, wherever bnode type-specific operations are required. Note that the only member of the `struct bnode_ops` function array that does *not* have a corresponding invocation macro defined is *create()*, which is always called directly.

3.5.1 *BOP_TIMEOUT()*

```
#define BOP_TIMEOUT(bnode) \
    ((*(bnode)->ops->timeout)((bnode)))
```

Execute the bnode type-specific actions required when a timeout action must be taken. This macro takes a single argument, namely a pointer to a type-specific bnode structure.

3.5.2 *BOP_GETSTAT()*

```
#define BOP_GETSTAT(bnode, a) \
    ((*(bnode)->ops->getstat)((bnode),(a)))
```

Execute the bnode type-specific actions required when a caller is attempting to get status information concerning the bnode. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a pointer to a longword in which the desired status value will be placed.

3.5.3 *BOP_SETSTAT()*

```
#define BOP_SETSTAT(bnode, a) \
    ((*(bnode)->ops->setstat)((bnode),(a)))
```

Execute the bnode type-specific actions required when a caller is attempting to set the status information concerning the bnode. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a longword from which the new status value is obtained.

3.5.4 *BOP_DELETE()*

```
#define BOP_DELETE(bnode) \
    ((*(bnode)->ops->delete)((bnode)))
```

Execute the bnode type-specific actions required when a bnode is deleted. This macro takes a single argument, namely a pointer to a type-specific bnode structure.

3.5.5 *BOP_PROCEXIT()*

```
#define BOP_PROCEXIT(bnode, a) \
    ((*(bnode)->ops->procexit)((bnode),(a)))
```

Execute the bnode type-specific actions required whenever the UNIX process implementing the given bnode exits. It takes two parameters, the first being a pointer to a type-specific bnode structure, and the second being a pointer to the `struct bnode_proc` (defined in Section 3.3.9), describing that process in detail.

3.5.6 *BOP_GETSTRING()*

```
#define BOP_GETSTRING(bnode, a, b) \
    ((*(bnode)->ops->getstring)((bnode),(a), (b)))
```

Execute the bnode type-specific actions required when the status string for the given bnode must be fetched. It takes three parameters. The first is a pointer to a type-specific bnode structure, the second is a pointer to a character buffer, and the third is a longword specifying the size, in bytes, of the above buffer.

3.5.7 *BOP_GETPARM()*

```
#define BOP_GETPARM(bnode, n, b, l) \
    ((*(bnode)->ops->getparm)((bnode),(n),(b),(l)))
```

Execute the bnode type-specific actions required when a particular parameter string for the given bnode must be fetched. It takes four parameters. The first is a pointer to a type-specific bnode structure, the second is a longword identifying the index of the desired parameter string, the third is a pointer to a character buffer to receive the parameter string, and the fourth and final argument is a longword specifying the size, in bytes, of the above buffer.

3.5.8 *BOP_RESTARTP()*

```
#define BOP_RESTARTP(bnode) \
    ((*(bnode)->ops->restartp)((bnode)))
```

Execute the bnode type-specific actions required when the UNIX process implementing the bnode of this type is restarted. It is expected that the stored process command line will be parsed in preparation for the coming execution. It takes a single argument, a pointer to a type-specific bnode structure from which the command line can be located.

3.5.9 *BOP_HASCORE()*

```
#define BOP_HASCORE(bnode)  ((*(bnode)->ops->hascore)((bnode)))
```

Execute the bnode type-specific actions required when it must be determined whether or not the attached process currently has a stored core file. It takes a single argument, a pointer to a type-specific bnode structure from which the name of the core file may be constructed.

3.6 Functions

This section covers the *BOS Server* RPC interface routines. They are generated from the *bosint.xg Rxgen* file. At a high level, these functions may be seen as belonging to seven basic classes:

- Creating and removing process entries
- Examining process information
- Starting, stopping, and restarting processes
- Security configuration

- Cell configuration
- Installing/uninstalling server binaries
- Executing commands at the server

The following is a summary of the interface functions and their purpose, divided according to the above classifications:

Creating & Removing Process Entries	
Function Name	**Description**
BOZO_CreateBnode()	Create a process instance.
BOZO_DeleteBnode()	Delete a process instance.

Examining Process Information	
Function Name	**Description**
BOZO_GetStatus()	Get status information for the given process instance.
BOZO_EnumerateInstance()	Get instance name from the *i'th* bnode.
BOZO_GetInstanceInfo()	Get information on the given process instance.
BOZO_GetInstanceParm()	Get text of command line associated with the given process instance.
BOZO_GetRestartTime()	Get one of the *BOS Server* restart times.
BOZO_SetRestartTime()	Set one of the *BOS Server* restart times.
BOZO_GetDates()	Get the modification times for versions of a server binary file.
StartBOZO_GetLog()	Pass the `IN` params when fetching a *BOS Server* log file.
EndBOZO_GetLog()	Get the `OUT` params when fetching a *BOS Server* log file.
BOZO_GetInstanceStrings()	Get strings related to a given process instance.

Starting, Stopping & Restarting Processes	
Function Name	**Description**
BOZO_SetStatus()	Set process instance status and goal.
BOZO_SetTStatus()	Temporarily set process instance status and goal.
BOZO_StartupAll()	Start all existing process instances.
BOZO_ShutdownAll()	Shut down all process instances.
BOZO_RestartAll()	Shut down, then restart all process instances.
BOZO_ReBozo()	Shut down, then restart all process instances and the *BOS Server* itself.
BOZO_Restart()	Restart a given process instance.
BOZO_WaitAll()	Wait until all process instances have reached their goals.

Security Configuration	
Function Name	**Description**
BOZO_AddSUser()	Add a user to the *UserList*.
BOZO_DeleteSUser()	Delete a user from the *UserList*.
BOZO_ListSUsers()	Get the name of the user in the given position in the *UserList* file.
BOZO_ListKeys()	List info about the key at a given index in the key file.
BOZO_AddKey()	Add a key to the key file.
BOZO_DeleteKey()	Delete the entry for an AFS key.
BOZO_SetNoAuthFlag()	Enable or disable authenticated call requirements.

Cell Configuration	
Function Name	**Description**
BOZO_GetCellName()	Get the name of the cell to which the *BOS Server* belongs.
BOZO_SetCellName()	Set the name of the cell to which the *BOS Server* belongs.
BOZO_GetCellHost()	Get the name of a database host given its index.
BOZO_AddCellHost()	Add an entry to the list of database server hosts.
BOZO_DeleteCellHost()	Delete an entry from the list of database server hosts.

Installing/Uninstalling Server Binaries	
Function Name	**Description**
StartBOZO_Install()	Pass the `IN` params when installing a server binary.
EndBOZO_Install()	Get the `OUT` params when installing a server binary.
BOZO_UnInstall()	Roll back from a server binary installation.
BOZO_Prune()	Throw away old versions of server binaries and core files.

Executing Commands at the Server	
Function Name	**Description**
BOZO_Exec()	Execute a shell command at the server.

All of the string parameters in these functions are expected to point to character buffers that are at least `BOZO_BSSIZE` long.

3.6.1 Creating and Removing Processes

The two interface routines defined in this section are used for creating and deleting bnodes, thus determining which processe instances the *BOS Server* must manage.

3.6.1.1 BOZO_CreateBnode — Create a process instance

```
int BOZO_CreateBnode(IN struct rx_connection *z_conn,
                     IN char *type,
                     IN char *instance,
                     IN char *p1,
                     IN char *p2,
                     IN char *p3,
                     IN char *p4,
                     IN char *p5,
                     IN char *p6)
```

Description

This interface function allows the caller to create a bnode (process instance) on the server machine executing the routine.

The instance's type is declared to be the string referenced in the `type` argument. There are three supported instance type names, namely `simple`, `fs`, and `cron` (see Section 2.1 for a detailed examination of the types of bnodes available).

The bnode's name is specified via the `instance` parameter. Any name may be chosen for a *BOS Server* instance. However, it is advisable to choose a name related to the name of the actual binary being instantiated. There are eight well-known names already in common use, corresponding to the ASF system agents. They are as follows:

- *kaserver* for the *Authentication Server.*
- *runntp* for the Network Time Protocol Daemon (*ntpd*).
- *ptserver* for the *Protection Server.*
- *upclient* for the client portion of the *UpdateServer*, which brings over binary files from */usr/afs/bin* directory and configuration files from */usr/afs/etc* directory on the system control machine.
- *upclientbin* for the client portion of the *UpdateServer*, which uses the */usr/afs/bin* directory on the binary distribution machine for this platform's CPU/operating system type.

- *upclientetc* for the client portion of the *UpdateServer*, which references the */usr/afs/etc* directory on the system control machine.
- *upserver* for the server portion of the *UpdateServer*.
- *vlserver* for the *Volume Location Server*.

Up to six command-line strings may be communicated in this routine, residing in arguments `p1` through `p6`. Different types of bnodes allow for different numbers of actual server processes to be started, and the command lines required for such instantiation are passed in this manner.

The given bnode's *setstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* macro.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to create new instances. If successfully created, the new *BOS Server* instance will be appended to the *BosConfig* file kept on the machine's local disk. The *UserList* and *BosConfig* files are examined in detail in Sections 2.3.1 and 2.3.4 respectively.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZEXISTS` The given instance already exists.

`BZBADTYPE` Illegal value provided in the `type` parameter.

`BZNOCREATE` Failed to create desired entry.

3.6.1.2 BOZO_DeleteBnode — Delete a process instance

```
int BOZO_DeleteBnode(IN struct rx_connection *z_conn,
                     IN char *instance)
```

Description

This routine deletes the *BOS Server* bnode whose name is specified by the `instance` parameter. If an instance with that name does not exist, this operation fails. Similarly, if the process or processes associated with the given bnode have not been shut down (see the descriptions for the *BOZO_ShutdownAll()* and *BOZO_ShutdownAll()* interface functions), the operation also fails.

The given bnode's *setstat()* and *delete()* routines from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* and *BOP_DELETE()* macros.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to delete existing instances. If successfully deleted, the old *BOS Server* instance will be removed from the *BosConfig* file kept on the machine's local disk.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZNOENT` The given `instance` name not registered with the *BOS Server*.

`BZBUSY` The process(es) associated with the given `instance` are still active (i.e., a shutdown has not yet been performed or has not yet completed).

3.6.2 Examining Process Information

This section describes the ten interface functions that collectively allow callers to obtain and modify the information stored by the *BOS Server* to describe the set of process that it manages. Among the operations supported by the functions examined here are getting and setting status information, obtaining the instance parameters, times, and dates, and getting the text of log files on the server machine

3.6.2.1 **BOZO_GetStatus** — Get status information for the given process instance

```
int BOZO_GetStatus(IN struct rx_connection *z_conn,
                   IN char *instance,
                   OUT long *intStat,
                   OUT char **statdescr)
```

Description

This interface function looks up the bnode for the given process `instance` and places its numerical status indicator into `intStat` and its status string (if any) into a buffer referenced by `statdescr`.

The set of values that may be returned in the `intStat` argument are defined fully in Section 3.2.3. Briefly, they are `BSTAT_STARTINGUP`, `BSTAT_NORMAL`, `BSTAT_SHUTTINGDOWN`, and `BSTAT_SHUTDOWN`.

A buffer holding `BOZO_BSSIZE` (256) characters is allocated, and `statdescr` is set to point to it. Not all bnodes types implement status strings, which are used to provide additional status information for the class. An example of one bnode type that does define these strings is `fs`, which exports the following status strings:

- `"file server running"`
- `"file server up; volser down"`
- `"salvaging file system"`
- `"starting file server"`
- `"file server shutting down"`
- `"salvager shutting down"`
- `"file server shut down"`

The given bnode's *getstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_GETSTAT()* macro.

Error Codes

`BZNOENT` The given process `instance` is not registered with the *BOS Server*.

3.6.2.2 BOZO_EnumerateInstance — Get instance name from *i'th* bnode

```
int BOZO_EnumerateInstance(IN struct rx_connection *z_conn,
                           IN long instance,
                           OUT char **iname);
```

Description

This routine will find the bnode describing process instance number `instance` and return that instance's name in the buffer to which the `iname` parameter points. This function is meant to be used to enumerate all process instances at a *BOS Server*. The first legal `instance` number value is zero, which will return the instance name from the first registered bnode. Successive values for `instance` will return information from successive bnodes. When all bnodes have been thus enumerated, the *BOZO_EnumerateInstance()* function will return `BZDOM`, indicating that the list of bnodes has been exhausted.

Error Codes

`BZDOM` The instance number indicated in the `instance` parameter does not exist.

3.6.2.3 BOZO_GetInstanceInfo — Get information on the given process instance

```
int BOZO_GetInstanceInfo(IN struct rx_connection *z_conn,
                         IN char *instance,
                         OUT char **type,
                         OUT struct bozo_status *status)
```

Description

Given the string name of a *BOS Server* `instance`, this interface function returns the `type` of the instance and its associated `status` descriptor. The set of values that may be placed into the `type` parameter are `simple`, `fs`, and `cron` (see Section 2.1 for a detailed examination of the types of bnodes available). The `status` structure filled in by the call includes such information as the goal and file goals, the process start time, the number of times the process has started, exit information, and whether or not the process has a core file.

Error Codes

`BZNOENT` The given process `instance` is not registered with the *BOS Server*.

3.6.2.4 BOZO_GetInstanceParm — Get text of command line associated with the given process instance

```
int BOZO_GetInstanceParm(IN struct rx_connection *z_conn,
                         IN char *instance,
                         IN long num,
                         OUT char **parm)
```

Description

Given the string name of a *BOS Server* process `instance` and an index identifying the associated command line of interest, this routine returns the text of the desired command line. The first associated command line text for the `instance` may be acquired by setting the index parameter, `num`, to zero. If an index is specified for which there is no matching command line stored in the bnode, then the function returns `BZDOM`.

Error Codes

`BZNOENT` The given process `instance` is not registered with the *BOS Server*.

`BZDOM` There is no command line text associated with index `num` for this bnode.

3.6.2.5 BOZO_GetRestartTime — Get one of the *BOS Server* restart times

```
int BOZO_GetRestartTime(IN struct rx_connection *z_conn,
                        IN long type,
                        OUT struct bozo_netKTime *restartTime)
```

Description

The *BOS Server* maintains two different restart times, for itself and all server processes it manages, as described in Section 2.4. Given which one of the two types of restart time is desired, this routine fetches the information from the *BOS Server*. The `type` argument is used to specify the exact restart time to fetch. If `type` is set to one (1), then the general restart time for all agents on the machine is fetched. If `type` is set to two (2), then the new-binary restart time is returned. A value other than these two for the `type` parameter results in a return value of `BZDOM`.

Error Codes

`BZDOM` All illegal value was passed in via the `type` parameter.

3.6.2.6 BOZO_SetRestartTime — Set one of the *BOS Server* restart times

```
int BOZO_SetRestartTime(IN struct rx_connection *z_conn,
                        IN long type,
                        IN struct bozo_netKTime *restartTime)
```

Description

This function is the inverse of the *BOZO_GetRestartTime()* interface routine described in Section 3.6.2.5 above. Given the `type` of restart time and its new value, this routine will set the desired restart time at the *BOS Server* receiving this call. The values for the `type` parameter are identical to those used by *BOZO_GetRestartTime()*, namely one (1) for the general restart time and two (2) for the new-binary restart time.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to set its restart times.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZDOM` All illegal value was passed in via the `type` parameter.

3.6.2.7 **BOZO_GetDates** — Get the modification times for versions of a server binary file

```
int BOZO_GetDates(IN struct rx_connection *z_conn,
                  IN char *path,
                  OUT long *newtime,
                  OUT long *baktime,
                  OUT long *oldtime)
```

Description

Given a fully-qualified pathname identifying the particular server binary to examine in the `path` argument, this interface routine returns the modification time of that file, along with the modification times for the intermediate (`.BAK`) and old (`.OLD`) versions. The above-mentioned times are deposited into the `newtime`, `baktime` and `oldtime` arguments. Any one or all of the reported times may be set to zero, indicating that the associated file does not exist.

Error Codes

--- None.

3.6.2.8 StartBOZO_GetLog — Pass the IN params when fetching a *BOS Server* log file

```
int BOZO_StartGetLog(IN struct rx_connection *z_conn,
                     IN char *name)
```

Description

The *BOZO_GetLog()* function defined in the *BOS Server Rxgen* interface file is used to acquire the contents of the given log file from the machine processing the call. It is defined to be a *streamed* function, namely one that can return an arbitrary amount of data. For full details on the definition and use of streamed functions, please refer to the *Streamed Function Calls* section in [4].

This function is created by *Rxgen* in response to the *BOZO_GetLog()* interface definition in the *bosint.xg* file. The *StartBOZO_GetLog()* routine handles passing the `IN` parameters of the streamed call to the *BOS Server*. Specifically, the `name` parameter is used to convey the string name of the desired log file. For the purposes of opening the specified files at the machine being contacted, the current working directory for the *BOS Server* is considered to be */usr/afs/logs*. If the caller is included in the locally-maintained *UserList* file, any pathname may be specified in the `name` parameter, and the contents of the given file will be fetched. All other callers must provide a string that does *not* include the slash character, as it might be used to construct an unauthorized request for a file outside the */usr/afs/logs* directory.

Error Codes

`RXGEN_CC_MARSHAL` The transmission of the *GetLog()* `IN` parameters failed. This and all *rxgen* constant definitions are available from the *rxgen_consts.h* include file.

3.6.2.9 EndBOZO_GetLog — Get the `OUT` params when fetching a *BOS Server* log file

`int` **BOZO_EndGetLog**`(IN struct rx_connection *z_conn)`

Description

This function is created by *Rxgen* in response to the *BOZO_GetLog()* interface definition in the *bosint.xg* file. The *EndBOZO_GetLog()* routine handles the recovery of the `OUT parameters` for this interface call (of which there are none). The utility of such functions is often the value they return. In this case, however, *EndBOZO_GetLog()* always returns success. Thus, it is not even necessary to invoke this particular function, as it is basically a no-op.

Error Codes

`---` Always returns successfully.

3.6.2.10 BOZO_GetInstanceStrings — Get strings related to a given process instance

```
int BOZO_GetInstanceStrings(IN struct rx_connection *z_conn,
                            IN char *instance,
                            OUT char **errorName,
                            OUT char **spare1,
                            OUT char **spare2,
                            OUT char **spare3)
```

Description

This interface function takes the string name of a *BOS Server* instance and returns a set of strings associated with it. At the current time, there is only one string of interest returned by this routine. Specifically, the `errorName` parameter is set to the error string associated with the bnode, if any. The other arguments, `spare1` through `spare3`, are set to the null string. Note that memory is allocated for all of the `OUT` parameters, so the caller should be careful to free them once it is done.

Error Codes

`BZNOENT` The given process `instance` is not registered with the *BOS Server*.

3.6.3 Starting, Stopping, and Restarting Processes

The eight interface functions described in this section allow *BOS Server* clients to manipulate the execution of the process instances the *BOS Server* controls.

3.6.3.1 BOZO_SetStatus — Set process instance status and goal

```
int BOZO_SetStatus(IN struct rx_connection *z_conn,
                   IN char *instance,
                   IN long status)
```

Description

This routine sets the actual status field, as well as the "file goal", of the given `instance` to the value supplied in the `status` parameter. Legal values for `status` are taken from the set described in Section 3.2.3, specifically `BSTAT_NORMAL` and `BSTAT_SHUTDOWN`. For more information about these constants (and about goals/file goals), please refer to Section 3.2.3.

The given bnode's *setstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* macro.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to perform this operation. If successfully modified, the *BOS Server* bnode defining the given instance will be written out to the *BosConfig* file kept on the machine's local disk.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZNOENT` The given `instance` name not registered with the *BOS Server*.

3.6.3.2 BOZO_SetTStatus — Temporarily set process instance status and goal

```
int BOZO_SetTStatus(IN struct rx_connection *z_conn,
                    IN char *instance,
                    IN long status)
```

Description

This interface routine is much like the *BOZO_SetStatus()*, defined in Section 3.6.3.1 above, except that its effect is to set the instance status on a temporary basis. Specifically, the status field is set to the given `status` value, but the "file goal" field is not changed. Thus, the instance's stated goal has not changed, just its current status.

The given bnode's *setstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* macro.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to perform this operation. If successfully modified, the *BOS Server* bnode defining the given instance will be written out to the *BosConfig* file kept on the machine's local disk.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZNOENT` The given `instance` name not registered with the *BOS Server*.

3.6.3.3 BOZO_StartupAll — Start all existing process instances

`int` **BOZO_StartupAll**`(IN struct rx_connection *z_conn)`

Description

This interface function examines all bnodes and attempts to restart all of those that have not been explicitly been marked with the `BSTAT_SHUTDOWN` file goal. Specifically, *BOP_SETSTAT()* is invoked, causing the *setstat()* routine from each bnode's *ops* array to be called. The bnode's `flags` field is left with the `BNODE_ERRORSTOP` bit turned off after this call.

The given bnode's *setstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* macro.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to start up bnode process instances.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

3.6.3.4 BOZO_ShutdownAll — Shut down all process instances

`int` **BOZO_ShutdownAll**`(IN struct rx_connection *z_conn)`

Description

This interface function iterates through all bnodes and attempts to shut them all down. Specifically, the *BOP_SETSTAT()* is invoked, causing the *setstat()* routine from each bnode's *ops* array to be called, setting that bnode's `goal` field to `BSTAT_SHUTDOWN`.

The given bnode's *setstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_SETSTAT()* macro.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to perform this operation.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

3.6.3.5 BOZO_RestartAll — Shut down, then restart all process instances

int **BOZO_RestartAll**(IN struct rx_connection *z_conn)

Description

This interface function shuts down every *BOS Server* process instance, waits until the shutdown is complete (i.e., all instances are registered as being in state BSTAT_SHUTDOWN), and then starts them all up again. While all the processes known to the *BOS Server* are thus restarted, the invocation of the *BOS Server* itself does *not* share this fate. For simulation of a truly complete machine restart, as is necessary when an far-reaching change to a database file has been made, use the *BOZO_ReBozo()* interface routine defined in Section 3.6.3.6 below.

The given bnode's *getstat()* and *setstat()* routines from its individual *ops* array will be called in the course of this execution via the *BOP_GETSTAT()* and *BOP_SETSTAT()* macros.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to restart bnode process instances.

Error Codes

BZACCESS The caller is not authorized to perform this operation.

3.6.3.6 BOZO_ReBozo — Shut down, then restart all process instances and the *BOS Server* itself

```
int BOZO_ReBozo(IN struct rx_connection *z_conn)
```

Description

This interface routine is identical to the *BOZO_RestartAll()* call, defined in Section 3.6.3.5 above, except for the fact that the *BOS Server* itself is restarted in addition to all the known bnodes. All of the *BOS Server*'s open file descriptors are closed, and the standard *BOS Server* binary image is started via *execve()*.

The given bnode's *getstat()* and *setstat()* routines from its individual *ops* array will be called in the course of this execution via the *BOP_GETSTAT()* and *BOP_SETSTAT()* macros.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to restart bnode process instances.

Error Codes

BZACCESS	The caller is not authorized to perform this operation.

3.6.3.7 BOZO_Restart — Restart a given process instance

```
int BOZO_Restart(IN struct rx_connection *z_conn,
                 IN char *instance)
```

Description

This interface function is used to shut down and then restart the process instance identified by the `instance` string passed as an argument.

The given bnode's *getstat()* and *setstat()* routines from its individual *ops* array will be called in the course of this execution via the *BOP_GETSTAT()* and *BOP_SETSTAT()* macros.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to restart bnode process instances.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZNOENT` The given `instance` name not registered with the *BOS Server*.

3.6.3.8 BOZO_WaitAll — Wait until all process instances have reached their goals

`int` **BOZO_WaitAll**`(IN struct rx_connection *z_conn)`

Description

This interface function is used to synchronize with the status of the bnodes managed by the *BOS Server*. Specifically, the *BOZO_WaitAll()* call returns when each bnode's current status matches the value in its short-term `goal` field. For each bnode it manages, the *BOS Server* thread handling this call invokes the *BOP_GETSTAT()* macro, waiting until the bnode's status and goals line up.

Typically, the *BOZO_WaitAll()* routine is used to allow a program to wait until all bnodes have terminated their execution (i.e., all `goal` fields have been set to `BSTAT_SHUTDOWN` and all corresponding processes have been killed). Note, however, that this routine may also be used to wait until all bnodes *start up*. The true utility of this application of *BOZO_WaitAll()* is more questionable, since it will return when all bnodes have simply *commenced execution*, which does not imply that they have completed their initialization phases and are thus rendering their normal services.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to wait on bnodes through this interface function.

The given bnode's *getstat()* routine from its individual *ops* array will be called in the course of this execution via the *BOP_GETSTAT()* macro.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

3.6.4 Security Configuration

This section describes the seven *BOS Server* interface functions that allow a properly-authorized person to examine and modify certain data relating to system security. Specifically, it allows for manipulation of the list of adminstratively "privileged" individuals, the set of Kerberos keys used for file service, and whether authenticated connections should be required by the *BOS Server* and all other AFS server agents running on the machine.

3.6.4.1 BOZO_AddSUser — Add a user to the *UserList*

```
int BOZO_AddSUser(IN struct rx_connection *z_conn,
                  IN char *name);
```

Description

This interface function is used to add the given user `name` to the *UserList* file of priviledged *BOS Server* principals. Only individuals already appearing in the *UserList* are permitted to add new entries. If the given user `name` already appears in the file, the function fails. Otherwise, the file is opened in append mode and the `name` is written at the end with a trailing newline.

Error Codes

- `BZACCESS` The caller is not authorized to perform this operation.
- `EEXIST` The individual specified by `name` is already on the *UserList*.
- `EIO` If the *UserList* file could not be opened or closed.

3.6.4.2 BOZO_DeleteSUser — Delete a user from the *UserList*

```
int BOZO_DeleteSUser(IN struct rx_connection *z_conn,
                     IN char *name)
```

Description

This interface function is used to delete the given user `name` from the *UserList* file of priviledged *BOS Server* principals. Only individuals already appearing in the *UserList* are permitted to delete existing entries. The file is opened in read mode, and a new file named *UserList.NXX* is created in the same directory and opened in write mode. The original *UserList* is scanned, with each entry copied to the new file if it doesn't match the given `name`. After the scan is done, all files are closed, and the *UserList.NXX* file is renamed to *UserList*, overwriting the original.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`-1` The *UserList* file could not be opened.

`EIO` The *UserList.NXX* file could not be opened, or an error occured in the file close operations.

`ENOENT` The given `name` was not found in the original *UserList* file.

3.6.4.3 BOZO_ListSUsers — Get the name of the user in the given position in the *UserList* file

```
int BOZO_ListSUsers(IN struct rx_connection *z_conn,
                    IN long an,
                    OUT char **name)
```

Description

This interface function is used to request the `name` of priviledged user in the `an`'th slot in the *BOS Server*'s *UserList* file. The string placed into the `name` parameter may be up to 256 characters long, including the trailing null.

Error Codes

1 The *UserList* file could not be opened, or an invalid value was specified for `an`.

3.6.4.4 BOZO_ListKeys — List info about the key at a given index in the key file

```
int BOZO_ListKeys(IN struct rx_connection *z_conn,
                  IN long an,
                  OUT long *kvno,
                  OUT struct bozo_key *key,
                  OUT struct bozo_keyInfo *keyinfo)
```

Description

This interface function allows its callers to specify the index of the desired AFS encryption key, and to fetch information regarding that key. If the caller is properly authorized, the version number of the specified key is placed into the `kvno` parameter. Similarly, a description of the given key is placed into the `keyinfo` parameter. When the *BOS Server* is running in noauth mode, the key itself will be copied into the `key` argument, otherwise the `key` structure will be zeroed. The data placed into the `keyinfo` argument, declared as a `struct bozo_keyInfo` as defined in Section 3.3.3, is obtained as follows. The `mod_sec` field is taken from the value of `st_mtime` after *stat()*ing */usr/afs/etc/KeyFile*, and the `mod_usec` field is zeroed. The *keyCheckSum* is computed by an *Authentication Server* routine, which calculates a 32-bit cryptographic checksum of the key, encrypting a block of zeros and then using the first 4 bytes as the checksum.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to obtain information regarding the list of AFS keys held by the given *BOS Server*.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`BZDOM` An invalid index was found in the `an` parameter.

`KABADKEY` Defined in the exported *kautils.h* header file corresponding to the *Authentication Server*, this return value indicates a problem with generating the checksum field of the `keyinfo` parameter.

3.6.4.5 BOZO_AddKey — Add a key to the key file

```
int BOZO_AddKey(IN struct rx_connection *z_conn,
                IN long an,
                IN struct bozo_key *key)
```

Description

This interface function allows a properly-authorized caller to set the value of key version number `an` to the given AFS `key`. If a slot is found in the key file */usr/afs/etc/KeyFile* marked as key version number `an`, its value is overwritten with the `key` provided. If an entry for the desired key version number does not exist, the key file is grown, and the new entry filled with the specified information.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to add new entries into the list of AFS keys held by the *BOS Server*.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`AFSCONF_FULL` The system key file already contains the maximum number of keys (`AFSCONF_MAXKEYS`, or 8). These two constant defintions are available from the *cellconfig.h* and *keys.h* AFS include files respectively.

3.6.4.6 **BOZO_DeleteKey** — Delete the entry for an AFS key

```
int BOZO_DeleteKey(IN struct rx_connection *z_conn,
                   IN long an)
```

Description

This interface function allows a properly-authorized caller to delete key version number `an` from the key file, */usr/afs/etc/KeyFile*. The existing keys are scanned, and if one with key version number `an` is found, it is removed. Any keys occurring after the deleted one are shifted to remove the file entry entirely.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to delete entries from the list of AFS keys held by the *BOS Server*.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`AFSCONF_NOTFOUND` An entry for key version number `an` was not found. This constant defintion is available from the *cellconfig.h* AFS include file.

3.6.4.7 BOZO_SetNoAuthFlag — Enable or disable requirement for authenticated calls

```
int BOZO_SetNoAuthFlag(IN struct rx_connection *z_conn,
                       IN long flag)
```

Description

This interface routine controls the level of authentication imposed on the *BOS Server* and all other AFS server agents on the machine by manipulating the *NoAuth* file in the */usr/afs/local* directory on the server. If the `flag` parameter is set to zero (0), the *NoAuth* file will be removed, instructing the *BOS Server* and AFS agents to authenenticate the RPCs they receive. Otherwise, the file is created as an indication to honor all RPC calls to the *BOS Server* and AFS agents, regardless of the credentials carried by callers.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

3.6.5 Cell Configuration

The five interface functions covered in this section all have to do with manipulating the configuration information of the machine on which the *BOS Server* runs. In particular, one may get and set the cell name for that server machine, enumerate the list of server machines running database servers for the cell, and add and delete machines from this list.

3.6.5.1 BOZO_GetCellName — Get the name of the cell to which the *BOS Server* belongs

```
int BOZO_GetCellName(IN struct rx_connection *z_conn,
                     OUT char **name)
```

Description

This interface routine returns the name of the cell to which the given *BOS Server* belongs. The *BOS Server* consults a file on its local disk, */usr/afs/etc/ThisCell* to obtain this information. If this file does not exist, then the *BOS Server* will return a null string.

Error Codes

`AFSCONF_UNKNOWN` The *BOS Server* could not access the cell name file. This constant defintion is available from the *cellconfig.h* AFS include file.

3.6.5.2 BOZO_SetCellName — Set the name of the cell to which the *BOS Server* belongs

```
int BOZO_SetCellName(IN struct rx_connection *z_conn,
                     IN char *name)
```

Description

This interface function allows the caller to set the name of the cell to which the given *BOS Server* belongs. The *BOS Server* writes this information to a file on its local disk, */usr/afs/etc/ThisCell*. The current contents of this file are first obtained, along with other information about the current cell. If this operation fails, then *BOZO_SetCellName()* also fails. The string `name` provided as an argument is then stored in *ThisCell*.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to set the name of the cell to which the machine executing the given *BOS Server* belongs.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`AFSCONF_NOTFOUND` Information about the current cell could not be obtained. This constant definition, along with `AFSCONF_FAILURE` appearing below, is availabel from the *cellconfig.h* AFS include file.

`AFSCONF_FAILURE` New cell name could not be written to file.

3.6.5.3 BOZO_GetCellHost — Get the name of a database host given its index

```
int BOZO_GetCellHost(IN struct rx_connection *z_conn,
                     IN long awhich,
                     OUT char **name)
```

Description

This interface routine allows the caller to get the name of the host appearing in position `awhich` in the list of hosts acting as database servers for the *BOS Server*'s cell. The first valid position in the list is index zero. The host's name is deposited in the character buffer pointed to by `name`. If the value of the index provided in `awhich` is out of range, the function fails and a null string is placed in `name`.

Error Codes

`BZDOM` The host index in `awhich` is out of range.

`AFSCONF_NOTFOUND` Information about the current cell could not be obtained. This constant defintion may be found in the *cellconfig.h* AFS include file.

3.6.5.4 **BOZO_AddCellHost** — Add an entry to the list of database server hosts

```
int BOZO_AddCellHost(IN struct rx_connection *z_conn,
                     IN char *name)
```

Description

This interface function allows properly-authorized callers to add a `name` to the list of hosts running AFS database server processes for the *BOS Server*'s home cell. If the given name does not already appear in the database server list, a new entry will be created. Regardless, the mapping from the given name to its IP address will be recomputed, and the cell database file, */usr/afs/etc/CellServDB* will be updated.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to add an entry to the list of host names providing database services for the *BOS Server*'s home cell.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`AFSCONF_NOTFOUND` Information about the current cell could not be obtained. This constant defintion may be found in the *cellconfig.h* AFS include file.

3.6.5.5 BOZO_DeleteCellHost — Delete an entry from the list of database server hosts

```
int BOZO_DeleteCellHost(IN struct rx_connection *z_conn,
                        IN char *name)
```

Description

This interface routine allows properly-authorized callers to remove a given `name` from the list of hosts running AFS database server processes for the *BOS Server*'s home cell. If the given name does not appear in the database server list, this function will fail. Otherwise, the matching entry will be removed, and the cell database file, */usr/afs/etc/CellServDB* will be updated.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to delete an entry from the list of host names providing database services for the *BOS Server*'s home cell.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`AFSCONF_NOTFOUND` Information about the current cell could not be obtained. This constant defintion may be found in the *cellconfig.h* AFS include file.

3.6.6 Installing/Uninstalling Server Binaries

There are four *BOS Server* interface routines that allow administrators to install new server binaries and to roll back to older, perhaps more reliable, executables. They also allow for stored images of the old binaries (as well as core files) to be "pruned", or selectively deleted.

3.6.6.1 StartBOZO_Install — Pass the IN params when installing a server binary

```
int StartBOZO_Install(IN struct rx_connection *z_conn,
                      IN char *path,
                      IN long size,
                      IN long flags,
                      IN long date)
```

Description

The *BOZO_Install()* function defined in the *BOS Server Rxgen* interface file is used to deliver the executable image of an AFS server process to the given server machine and then installing it in the appropriate directory there. It is defined to be a *streamed* function, namely one that can deliver an arbitrary amount of data. For full details on the definition and use of streamed functions, please refer to the *Streamed Function Calls* section in [4].

This function is created by *Rxgen* in response to the *BOZO_Install()* interface definition in the *bosint.xg* file. The *StartBOZO_Install()* routine handles passing the `IN` parameters of the streamed call to the *BOS Server*. Specifically, the `apath` argument specifies the name of the server binary to be installed (including the full pathname prefix, if necessary). Also, the length of the binary is communicated via the `size` argument, and the modification time the caller wants the given file to carry is placed in `date`. The `flags` argument is currently ignored by the *BOS Server*.

After the above parameters are delivered with *StartBOZO_Install()*, the *BOS Server* creates a file with the name given in the `path` parameter followed by a *.NEW* postfix. The `size` bytes comprising the text of the executable in question are then read over the RPC channel and stuffed into this new file. When the transfer is complete, the file is closed. The existing versions of the server binary are then "demoted"; the **.BAK* version (if it exists) is renamed to **.OLD*. overwriting the existing **.OLD* version if and only if an **.OLD* version does not exist, or if a **.OLD* exists and the .BAK file is at least seven days old. The main binary is then renamed to **.BAK*. Finally, the **.NEW* file is renamed to be the new standard binary image to run, and its modification time is set to `date`.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to install server software onto the machine on which the *BOS Server* runs.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

`100` An error was encountered when writing the binary image to the local disk file. The truncated file was closed and deleted on the *BOS Server*.

`101` More than `size` bytes were delivered to the *BOS Server* in the RPC transfer of the executable image.

`102` Fewer than `size` bytes were delivered to the *BOS Server* in the RPC transfer of the executable image.

3.6.6.2 **EndBOZO_Install** — Get the `OUT` params when installing a server binary

`int` **EndBOZO_Install**`(IN struct rx_connection *z_conn)`

Description

This function is created by *Rxgen* in response to the *BOZO_Install()* interface definition in the *bosint.xg* file. The *EndBOZO_Install()* routine handles the recovery of the `OUT parameters` for this interface call, of which there are none. The utility of such functions is often the value they return. In this case, however, *EndBOZO_Install()* always returns success. Thus, it is not even necessary to invoke this particular function, as it is basically a no-op.

Error Codes

`---` Always returns successfully.

3.6.6.3 BOZO_UnInstall — Roll back from a server binary installation

```
int BOZO_UnInstall(IN struct rx_connection *z_conn,
                   IN char *path)
```

Description

This interface function allows a properly-authorized caller to "roll back" from the installation of a server binary. If the **.BAK* version of the server named `path` exists, it will be renamed to be the main executable file. In this case, the **.OLD* version, if it exists, will be renamed to **.BAK*. If a **.BAK* version of the binary in question is not found, the **.OLD* version is renamed as the new standard binary file. If neither a **.BAK* or a **.OLD* version of the executable can be found, the function fails, returning the low-level UNIX error generated at the server.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to roll back server software on the machine on which the *BOS Server* runs.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

3.6.6.4 BOZO_Prune — Throw away old versions of server binaries and core files

```
int BOZO_Prune(IN struct rx_connection *z_conn,
               IN long flags)
```

Description

This interface routine allows a properly-authorized caller to prune the saved versions of server binaries resident on the machine on which the *BOS Server* runs. The */usr/afs/bin* directory on the server machine is scanned in directory order. If the `BOZO_PRUNEOLD` bit is set in the `flags` argument, every file with the **.OLD* extension is deleted. If the `BOZO_PRUNEBAK` bit is set in the `flags` argument, every file with the **.BAK* extension is deleted. Next, the */usr/afs/logs* directory is scanned in directory order. If the `BOZO_PRUNECORE` bit is set in the `flags` argument, every file with a name beginning with the prefix *core* is deleted.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to prune server software binary versions and core files on the machine on which the *BOS Server* runs.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

3.6.7 Executing Commands at the Server

There is a single interface function defined by the *BOS Server* that allows execution of arbitrary programs or scripts on any server machine on which a *BOS Server* process is active.

3.6.7.1 **BOZO_Exec** — Execute a shell command at the server

```
int BOZO_Exec(IN struct rx_connection *z_conn,
              IN char *cmd)
```

Description

This interface routine allows a properly-authorized caller to execute any desired shell command on the server on which the given *BOS Server* runs. There is currently no provision made to pipe the output of the given command's execution back to the caller through the RPC channel.

The *BOS Server* will only allow individuals listed in its locally-maintained *UserList* file to execute arbitrary shell commands on the server machine on which the *BOS Server* runs via this call.

Error Codes

`BZACCESS` The caller is not authorized to perform this operation.

Bibliography

[1] CMU Information Technology Center. *Synchronization and Caching Issues in the Andrew File System*, USENIX Proceedings, Dallas, TX, Winter 1988.

[2] Transarc Corporation. *AFS 3.0 Command Reference Manual*, F-30-0-D103, Pittsburgh, PA, April 1990.

[3] Zayas, Edward R., Transarc Corporation. *AFS-3 Programmer's Reference: Specification for the Rx Remote Procedure Call Facility*, FS-00-D164, Pittsburgh, PA, April 1991.

[4] Zayas, Edward R., Transarc Corporation. *AFS-3 Programmer's Reference: File Server/Cache Manager Interface*, FS-00-D162, Pittsburgh, PA, April 1991.

[5] Transarc Corporation. *AFS 3.0 System Administrator's Guide*, F-30-0-D102, Pittsburgh, PA, April 1990.

[6] Kazar, Michael L., Information Technology Center, Carnegie Mellon University. *Ubik - A Library For Managing Ubiquitous Data*, ITCID, Pittsburgh, PA, Month, 1988.

[7] Kazar, Michael L., Information Technology Center, Carnegie Mellon University. *Quorum Completion*, ITCID, Pittsburgh, PA, Month, 1988.

[8] S. R. Kleinman. *Vnodes: An Architecture for Multiple file System Types in Sun UNIX*, Conference Proceedings, 1986 Summer Usenix Technical Conference, pp. 238-247, El Toro, CA, 1986.

Index