Sysctl: A Distributed System Control Package

Salvatore DeSimone & Christine Lombardi
Project Agora
IBM T. J. Watson Research Center

Sysctl: A Distributed System Control Package

Salvatore DeSimone & Christine Lombardi – Project Agora, IBM T.J. Watson Research Center

ABSTRACT

The sysctl package is an authenticated client/server system for executing remote commands. It is conceptually similar to *rsh*, but adds Kerberos[1] authentication, an ACL-based command authorization mechanism, and a programmable Tcl-based[2] command language in its server.

The sysctl server component, **sysctld**, is a daemon that runs on all workstations. The client component lets users send sysctl commands to a **sysctld** server. If the user is authorized for the requested operation, the server executes it on behalf of the user and sends back the result. The operations sent by the user are processed at the server using a built-in command interpreter and can range from a single sysctl command to a complex sysctl script. The server has a multi-level authorization scheme to guard against unauthorized access to commands.

The sysctl server uses the embeddable command language Extended Tcl[3] as the foundation for its built-in interpreter. The server can dynamically link in external shell commands and Tcl procedures to integrate existing management tools or create new global or service-specific commands. Once a command is created inside a server's interpreter, it is accessible to any authorized user from any workstation.

Sysctl uses the Kerberos authentication service for reliable third-party authentication, a prerequisite for authorization checking in a distributed computing environment. The server's built-in authorization mechanism provides granularity down to the individual command level.

Introduction

The motivation for the sysctl package comes from the experience of managing hundreds of workstations. The Agora Project manages approximately 1000 workstations of various UNIX flavors at the IBM T.J. Watson Research Center. There are many software packages and services available to help in managing such an environment. At Agora, these include Kerberos[1] authentication, the Hesiod[4] name service, NFS and AFS file service, software maintenance programs such as SUP[5], and others. What is missing is a general purpose *glue* package that can integrate these pieces into a coherent, manageable system and fill the considerable gaps between them.

Problem Specification

Before discussing the details of the sysctl package and its operation, we present a list of the key problems it is designed to solve.

Security

The management tools used to administer a large site need to be secure not only to prevent outside intrusion, but also to prevent accidental disasters such as the inadvertent deletion of a critical data file. The standard UNIX security mechanisms were designed for a single system and are not sufficiently robust to provide security in a distributed computing environment. Kerberos provides a means of authenticating users in a networked environment, but the interface for using it is via C library routines. Most administrative scripts in Agora are written in perl[6], and there is little desire to re-write them in C to ‘‘kerberize’’ them.

Authorization

The term *authorization* is used here in a generic sense and refers to the ability of a central managing organization to assign a set of administrative privileges to a user. These privileges must be valid for a set of machines without giving that user administrative access to other parts of the environment. The inability to *partition* administrative privileges has been a problem at Agora. Many tools require *superuser* access on a particular machine, others use simple authorization methods based on host names or UNIX user names. Some systems, such as AFS, have advanced authorization schemes but do not provide the desired granularity. In most cases, the situation is *all-or-nothing*: a user has either no privileges or all privileges.

Specialized Services

A site that provides many services utilizes large numbers of servers with diverse management requirements. At Agora, we have found that frequently a separate script or daemon is created to maintain a particular service. For Agora, writing separate daemons from scratch is inefficient, especially when features such as Kerberos authentication require the code to be written in C. More importantly, writing separate daemons in the absence of an organized *methodology* has led to a splintering of the management tools. The interfaces to the servers are different, the methods for maintaining the servers are different, and authorization is usually omitted or done on an ad-hoc basis.

Location Independence

Many administrative tasks require the user to login to a particular machine to perform a function. In a distributed environment, the administrator should be able to issue commands from any workstation to any workstation.

Design Overview

The sysctl package consists of the following elements:

- Server program (**sysctld**) that runs on all workstations.
- Commands built-in to the server which form the base of the system control language.
- Configuration files that control aspects of the server operation as well as extend the command set available on a given machine.
- Client library (**libsysctl.a**) that contains a **sysctl()** function, providing a C language interface for communicating with **sysctld**.
- Client shell program (**sysctl**) which offers a command-line interface for communicating with **sysctld**.

Sysctl uses a client/server, Remote Procedure Call (RPC) model. The **sysctld** server is a multitasking server that runs on every workstation. Clients send commands to a server using either the client library routines or the shell command.

When the server receives a request, it uses Kerberos in combination with authorization lists to assign an *authorization level* to the client. The operation contained in the request is then passed to the built-in command interpreter for execution. The availability of built-in and external commands varies based on the authorization level of the client. Upon completion of the operation, the server sends back the result of the operation along with any output (**stdout** and **stderr**) generated in the process.

The server's built-in command language provides the building blocks for developing high-level administrative tasks. The ability to easily add new commands to the server's interpreter lets administrators create their own tasks tailored for their particular installation, and assign specific lists of users who are authorized to run them. Once a command is defined in a sysctl server, it is accessible to any authorized user from any workstation.

The Tcl Command Language

Sysctl uses the Tcl[2][3] embeddable command language as the foundation for its built-in interpreter. The Tcl library contains a parser for a simple command language and a collection of built-in utility commands. It also has a C interface that host programs can use to augment the built-in set of Tcl commands with application specific commands. The use of an embedded language, and Tcl in particular, provides several advantages:

- A command language interface is ideal for administrative tasks.
- The inherent extendibility of the Tcl interpreter allows the sysctl language to be extended without recompiling C code.
- The use of a command language lets clients send over *scripts*, rather than just discrete commands. This gives a high level of flexibility as new operations can be created dynamically by combining lower level commands into high-level tasks without having to pre-distribute or pre-register scripts.
- The use of an embedded language gives the sysctl server complete control over the set of commands available to a client. This control forms the basis of the server's multi-level authorization scheme.

Authorization

One of the traditional problems with UNIX administrative tools is that while they may have useful features, they usually don't have robust authorization mechanisms. At Project Agora, the goal is to develop tools and procedures for administering multi-campus, enterprise-wide environments. It is in these large environments where security and authorization become particularly relevant. The sysctl package's authorization scheme was designed specifically to solve the *all-or-nothing* authorization syndrome of programs like *rsh*(1) and others.

Kerberos and Access Control Lists

An authorization scheme needs a reliable *authentication* service in order to confirm the identity of a user. The sysctl package uses the Kerberos authentication service to provide trusted third-party authentication of users. The sysctl request sent from the client contains a Kerberos ticket that uniquely identifies the user that initiated the request. Given an authenticated identity, the server uses Access Control Lists (ACLs) to determine authorizations. Access Control Lists serve two purposes within the server:

- Identifying to the server its set of trusted users.
- Controlling access to sysctl commands.

A sysctl ACL is a plain text file. Each line of an ACL file is interpreted as either a Kerberos principal name or the name of another ACL. When an

ACL is searched for a principal name, files listed within the ACL are also checked. This hierarchical structure makes it easier to maintain customized sets of authorized users on certain machines while still maintaining a global default set of authorized users. For example, an ACL stored in AFS can contain a core set of system administrators and also the name of a local file that can be customized per machine.

Authorization Levels

Each sysctl server has one ACL that lists its trusted users. This ACL is used to determine a client's *base authorization level*. When a sysctl request is received, the server assigns it one of three authorization levels:

- **Unauthentic**. No ticket was sent in the request or the identity of the client was not verified via Kerberos.
- **Authentic**. The client has been authenticated via Kerberos but is not listed in the server's ACL of trusted users.
- **Trusted**. The client is authenticated and is listed in the server's ACL of trusted users.

The server internally manages three separate interpreters that map to the three base authorization levels. These interpreters are configured to provide access to those commands authorized for its level only. When a request is received and the base authorization level of the requester is determined, the commands contained in the request are passed to the appropriate interpreter for execution. Thus, the set of commands available to a user at a particular authorization level are restricted to the commands authorized for the associated interpreter.

Task Authorization

While the server's authorization levels prevent unauthorized users from directly executing certain sysctl commands, it is sometimes desirable to authorize users to run a *task* that may contain commands they are not ordinarily able to run. This task authorization is accomplished by defining external procedures.

When an external procedure is defined in the server it is assigned an authorization. This can be either one of the three base authorization levels or a separate ACL.

External procedures can contain commands that are not directly accessible to a user. For example, it is possible to create a procedure which any unauthenticated user can run that contains the Tcl **exec** command. When an unauthenticated user runs the procedure, the **exec** contained in it will run successfully even though that user is not authorized to run **exec** directly.

Authorization Variables

Prior to executing the client request, the server sets several read-only variables within the interpreters that provide information on the authenticated identity of the client. These variables are mirrored with environment variables giving external scripts access to this information.

The purpose of the authorization variables is to provide a mechanism for external commands and procedures to create their own authorization checking mechanism. This is useful in cases where the desired authorization strategy does not fit an ACL structure. For instance, the authorization for a password changing command might be that a user is only authorized to change his own password. The variables provide the elements needed to perform this type of check.

Programming the Sysctl Server

One of the distinguishing features of the sysctl package is the ability to program the server component. In a sense, sysctl can be thought of as a general-purpose, authenticated client/server system that an administrator can *program* to perform whatever functions are required to manage the environment. In many cases, using sysctl is preferable to RPC programming because with sysctl, the programming is achieved using external configuration files rather than C programming.

The /etc/sysctl.conf File

The **/etc/sysctl.conf** file is the sysctl server configuration file. It is used to specify additional commands and procedures to be made available through the server and to override default configuration values. The configuration file modifies the state of the server either by assigning values to configuration variables or by executing one or more configuration commands.

The configuration commands create read-only variables, register external commands and procedures, include other configuration files, and create *classes* of external commands. The syntax of these commands is shown below.

```
include filename
create var variable value
create cmd name auth yes|no command
create proc name auth args body
create class label filename
```

Assigning values to the variables **ACL**, **LOG**, and **KEY** define the server's ACL file, the log file, and the file containing the server's Kerberos key, respectively. Environment variables, such as the default **PATH**, are set by assigning values to the **env()** array.

The include Command

The **include** command specifies additional configuration files to be read by the server. This makes it easier to manage large numbers of external command definitions by allowing them to be split among a hierarchical set of files.

The create var Command

The **create var** command defines read-only variables in each of the three interpreters. Variables defined in the configuration file using the standard **set** command will only exist while the configuration files are being read.

The create cmd Command

The **create cmd** command makes external shell commands available as sysctl commands through the server. Four arguments are required for this command:

- **name**: The name by which the command is created inside the interpreters.
- **auth**: An authorization specifier for the command. This can be **NOAUTH**, **AUTH**, or **ACL** to assign it one of the built-in authorization levels, or it can be the path name of an ACL file.
- **yes|no**: Indicates whether or not command arguments from the client request are to be passed to the shell command. The parameters passed to the shell command are not interpreted directly by the server but special shell characters (e.g., *;|()[‘’]$<>\n&+*, etc.) are not allowed in any of the parameters. If any of these characters are found the command is rejected.
- **command**: The shell command(s) to execute. This can contain any valid shell (**/bin/sh**) syntax, including multi-line commands.

You can think of external commands as embedded shell scripts in the server. The following example defines a sysctl command to clean out the **/tmp** directory. Any authenticated user is allowed to run this command and no arguments will be passed to the shell invocation.

```
create cmd clean_tmp AUTH no {
  find /tmp -type f \
      -mtime +2 -atime +2 \
        | xargs rm -f
}
```

The create proc Command

The **create proc** command defines external sysctl procedures. This command requires four arguments:

- **name:** The name by which the command is created inside the interpreters.

```
set dir /vol/sysctl/src/sample/files

# These declarations override the default locations
set ACL /etc/sysctl.acl
set LOG /usr/adm/sysctl.log
set KEY /etc/srvtab

# Set an environment variable
set env(PATH) /bin:/usr/bin:/usr/ucb:/etc:/usr/etc

# These variables are set in the service interpreters and marked readonly
create var ARCH [exec /bin/arch]
create var VERSION [exec /etc/agora/version]

# Include command class files
create class sample $dir/sample.cmds

# Create a command
create cmd date NOAUTH no {
     /bin/date
}
# Create a procedure
create proc sysinfo AUTH {} {
     global SCLHOST ARCH VERSION
     echo $SCLHOST $ARCH $VERSION
}

# Check for local configurations
if [file exists /etc/sysctl.conf.local] {
     include /etc/sysctl.conf.local
}
```

Figure 1: Sample **/etc/sysctl.conf** File

- **auth:** An authorization specifier for the command. This can be **NOAUTH**, **AUTH**, or **ACL** to assign it one of the built-in authorization levels, or it can be the path name of an ACL file.
- **args:** A list of names of arguments to the procedure.
- **body:** A sysctl (Tcl) expression that forms the body of the new procedure.

Executing a sysctl procedure is usually more efficient than executing an external shell command since the server does not have to *exec* (3) a shell to execute the command. This example defines a procedure to report filesystem usage. Only users listed in the ACL **/etc/fs.acl** are authorized to run this command. The name of the filesystem to check is passed as an argument. The variable **SCLHOST** is one of the authorization variables set by the server.

```
create proc fsinfo /etc/fs.acl {fs} {
    global SCLHOST
    statfs $fs buf
    echo "Filesystem $fs on $SCLHOST:"
    echo "Size: $buf(size) KB"
    echo "Used: $buf(used) KB"
    echo "Free: $buf(free) KB"
}
```

The create class Command

The **create class** command defines classes of commands within the server. Command classes provide a way to organize commands into logical groups for clarity. The **label** parameter specifies a *tag* that is prepended to all commands defined in the command file. If a class named **test** is created and the class file defines a procedure named **help**, the procedure is created in the interpreters as **test:help**.

Sysctl Client Library

The sysctl client library, **libsysctl.a**, provides a C language interface for communicating with **sysctld** servers. A full description of the programming interface is available in Appendix A. The main routine in the library is the **sysctl()** function:

```
#include <sysctl.h>
sc_result *sysctl(char *host,
                  char *op,
                  sc_control *cntl)
```

The **host** parameter is a pointer to a string that contains the name of the host to contact. The **op** parameter is a pointer to a string that contains the operation to run on the remote server. The **op** string can contain any sysctl expression and can be anything from a single command with no arguments to an entire script. The **sysctl()** routine does not interpret the **op** parameter. The **cntl** parameter is a pointer to an **sc_control** structure. This structure controls the communication mode between the client and server and is used to determine whether information is encrypted before it is transmitted, connection and operation timeouts, the port to use to connect to the remote server, etc.

Before sending the operation to the remote host, **sysctl()** attempts to obtain a Kerberos ticket for the service **rcmd.***hostname*, where *hostname* is the fully qualified hostname[1] of **host**. If a ticket cannot be obtained, the operation is sent to the server without any authentication information.

The result of the operation is returned via an **sc_result** structure. This structure contains the exit status of the remote operation as well as any output generated. See Appendix A for more details.

Sysctl Client Command

The sysctl client command provides a command-line interface to the sysctl package. All of the features of the **sysctl()** library routine are accessible via command line arguments. It is ideally suited for integrating sysctl calls into shell or perl scripts. A complete listing of the command options is given in Appendix C.

Integrating Existing Applications

One of the strengths of the sysctl package is the ease with which it integrates with existing tools. The following example illustrates the steps necessary to convert a typical administrative perl script designed for stand alone operation to work with sysctl.

The example script is called **chmbox** and is used to change a user's mail alias in the file **/usr/lib/aliases**. It takes a username and the new mail destination as arguments. Using sysctl, this same script can be used to let users update their mail alias remotely in a global alias file. The command only allows users to change their own mail alias. The host that houses the global alias file is called **admin**. Listed below are the steps required to set this up:

1. Modify **/etc/sysctl.conf** on **admin** to register the **chmbox** command.
2. Add authorization checks to the existing **chmbox** script.
3. Restart **sysctld** on **admin** to activate the command.

First, register the command. For this example, the entry is placed in **/etc/sysctl.conf**. If there were a suite of mail service commands, it would probably make sense to create a separate command class file.

```
sysctl -h admin 'confadd {
create cmd chmbox AUTH yes /etc/chmbox
}'
```

[1]Defined as the value returned by the *gethostbyname* (3) system call.

The **confadd** command is used to modify the contents of a server's configuration files. The **create cmd** command defines an external command named **chmbox**, and associates the script **/etc/chmbox** with it. The **AUTH** value in the authorization field ensures that only authenticated users can run this command. Additional authorization checks are performed within the modified **chmbox**. The **yes** value in the parameter field indicates that **/etc/chmbox** will accept arguments.

Next, add an authorization check to the existing **/etc/chmbox** script. This check is inserted after the arguments have been processed but before any files are modified. The check fails if the user name given in the command-line does not match the authenticated name retrieved from the sysctl environment variables:

```
# Assume the name of the alias to
# change is in $user
$ruser = $ENV{'SCUSER'};
$rrealm = $ENV{'SCREALM'};
$lrealm = $ENV{'SCLREALM'};

if (($user ne $ruser) ||
    ($rrealm ne $lrealm)) {
      die "Permission denied.\n";
}
```

Finally, restart the server on **admin** by issuing the command:

```
sysctl -h admin svcrestart
```

Users can now change their mailbox by issuing the command:

sysctl -h admin chmbox *user mbox*

Better yet, create an executable script that contains the following lines:

```
#!/usr/agora/bin/sysctl -r
#host#admin

if {$argc != 2} {
  error {Usage: chmbox <user> <mbox>}
}
set user $argv(1)
set mbox $argv(2)

if {[regexp {(.+)@(.+)} $mbox] != 1} {
  error {Alias must be user@host}
}
chmbox $user $mbox
```

The **-r** (replay) option to **sysctl** is used to send an entire script over to a server. It requires a file name argument. In this example, we take advantage of the **#!** specifer which executes **sysctl** with the replay file name argument (the script name) appended. The **#host#** syntax in line 2 of the script allows a default server to be set. Ordinarily, if no host is specified on the command line via the **-h** or **-c** flags, **sysctl** sends the operation to the local host. But if **-r** is used, **sysctl** looks at the first line of the replay file (unless it starts with **#!** in which case it looks at the second line) and checks for the keyword **#host#**. If the keyword appears, **sysctl** interprets the word that follows it as the name of a host and sends the script to that host. This in effect produces a *transportable* script! There is also support for a **#hesiod#** keyword which causes **sysctl** to query the Hesiod name server to find the name of a server to use and a **#cluster#** keyword which causes the script to be run on all machines in the specified cluster.

Another feature demonstrated in the script is the use of the **argc** and **argv()** variables. When run with -r, **sysctl** treats all extra command line arguments (i.e., those not associated with a flag) as arguments to pass to the replay file and sets the **argc** and **argv()** variables in much the same way as the *exec*(3) family of C routines.

Sysctl Applications

The method of writing a sysctl application generally consists of the following steps:

1. Examine what you are trying to do and divide it along client/server lines.
2. Define the client/server interface which consists of sysctl command names and their arguments, return values, and authorizations.
3. Code the client front end. This can be a fancy GUI, a perl or shell script, or even a simple sysctl script. The front end normally collects some information before making the appropriate sysctl call(s).
4. Code the server sysctl commands. For relatively small items, these are coded as external sysctl procedures. For more complex things, perl or shell scripts are written and inserted into the server as external commands or with ‘‘glue’’ procedures.

The rest of this section describes a **Home Directory Mover** sysctl application used at Agora. Agora currently supports user home directories located either in central AFS storage or on NFS-mountable volumes on a user's private workstation. The application, named **mvhomedir**, provides a seamless mechanism for Agora system administrators to move a home directory from its current location to a new location. The requirements for the design were:

- The **mvhomedir** command must be executable only by a subset of Agora system administrators.
- The administrators are not authorized to directly create, delete, or modify NFS or AFS volumes, or any data file that gets modified during the moving of a home directory, such as automounter maps.
- The administrators are authorized to transfer data between machines only for the purposes of moving a user's home directory.

The basic design of the application is shown in the accompanying diagram. The application defines two sysctl command classes: **mvh** and **mvh_server**. The **mvh** class contains one command, **mvh:mvhomedir**. This command accepts arguments that give the user name, the current location of the home directory, and the new location. Either location can be in AFS or NFS. The **mvh_server** class contains a number of commands that actually move the home directory.

A front end program queries the administrator for information and then runs the sysctl command **mvh:mvhomedir** to a secure server. The sysctl server on the secure system only executes the request if the issuer is on the **mvh** ACL. If the ACL check passes, the secure server runs two sysctl calls as itself[2] on behalf of the administrator. The first call (**mvh_server:nfstarfrom**) extracts the data in the user's current home directory, and the second (**mvh_server:nfschome** and **mvh_server:nfstarto**) creates the new home directory and unpacks the files. Both of these backend sysctl calls are run in *socket* mode to handle the bulk transfer of data. The source and destination hosts both check to make sure the principal executing the tar commands (in this case the secure server) is authorized by checking the **mvh_server** ACL. The **mvh_server** command class also has a suite of commands that perform the same functions for AFS: **mvh_server:afschome**, **mvh_server:afstarto**, and **mvh_server:afstarfrom**.

Note that the **mvh** and **mvh_server** command classes have separate ACLs associated with them. The only principal listed in the **mvh_server** ACL is **rcmd.***host*, where *host* is the hostname of the secure server. This means that although the administrators are authorized to run the high-level **mvhomedir** command, they do not have the ability to directly run the tar commands or the commands which create the user's home directory volume.

[2]It uses the **-l** flag to the sysctl client command.

The entire application consists of two sysctl command files that contain the application's sysctl command definitions (about 80 lines of code), a front end script that collects and sanity-checks information from the administrators and then issues the sysctl call (about 500 lines of perl code), and a set of back-end perl scripts to handle complex tasks such as creating AFS volumes and mount points, modifying administrative files, etc. (total of about 850 lines of perl code).

Future Directions

The sysctl client and server code have been compiled and run under AIX, HPUX, SunOS, and IRIX. A port of the client code has been made to a mainframe VM system. Porting of the client and server code to lower-end platforms such as OS/2 and DOS/Windows would increase the viability of sysctl for use in administering a truly heterogeneous environment.

There are two items which present some opportunity for improvement to sysctl. The first would be to remove the ACL processing code from the server and install it into a separate C library. This would allow the server and other applications to share the same basic ACL structure. It would also be advantageous to allow ACLs to be retrieved from authorization servers and include a more robust set of objects, such as IP addresses, etc.

Second, it may be worthwhile to port the client and server to use the secure RPC available in the OSF/DCE product. The ACL service provided as part of DCE may also be relevant to the first item above.

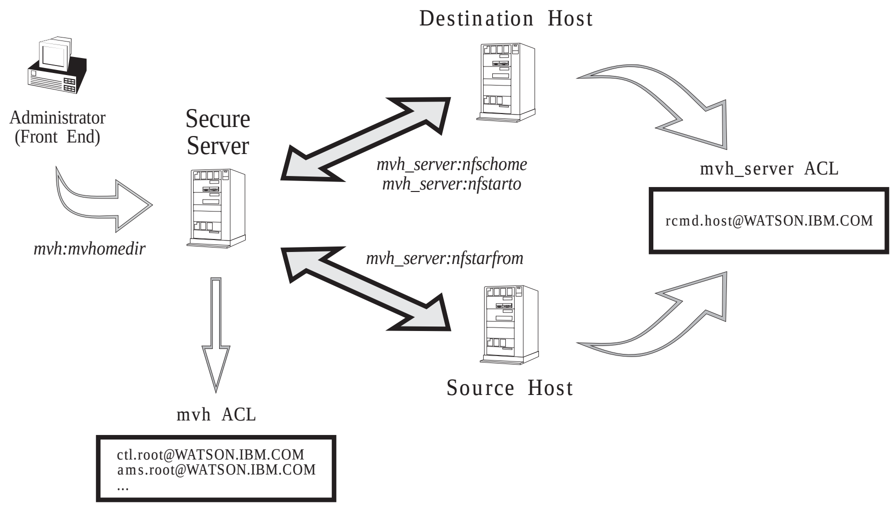

Figure 2: Home Directory Mover Design

Acknowledgements

The authors would like to acknowledge the contribution of Mike Dugan, who suggested rolling Tcl into the server. Also thanks to Mark Linehan, Aaron Shtromas, Jerry Larivee, and Lucien Van Elsen for their comments and suggestions on the design. The **Home Directory Mover** application was designed and implemented by Geetha Vijayan.

Availability

For information about the availability of **sysctl**, send mail to **agora-info@watson.ibm.com**.

Author Information

Salvatore DeSimone is a member of the System Architecture Group within Project Agora. Christine Lombardi is a member of the Agora Software Engineering Group. Their e-mail addresses are vatore@watson.ibm.com and ctl@watson.ibm.com. They can be reached via U.S. Mail at the IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY, 10532.

References

[1] Jennifer G. Steiner, Clifford Neuman, Jeffrey I. Schiller, *Kerberos: An Authentication Service for Open Network Systems*, Proc. USENIX Winter Conference, January 1988.

[2] John K. Ousterhout, *Tcl: An Embeddable Command Language*, Proc. USENIX Winter Conference, January 1990.

[3] Karl Lehenbauer, Mark Diekhans, *Extended Tcl – Extended Command Set for Tcl 6.2*, unpublished manual page, January 1992.

[4] Stephen P. Dyer, *The Hesiod Name Server*, Proc. USENIX Winter Conference, 1988.

[5] Stephen Shafer and Mary Thompson, *The SUP Software Upgrade Protocol*, Carnegie Mellon University, School of Computer Science, 1988.

[6] Larry Wall, Randal L. Schwartz, *Programming perl*, O'Reilly and Associates, Sebastopol, CA, 1990.

Appendix A – Programming Interface

```
#include <sysctl.h>
sc_result *sysctl(char *host,
                  char *op,
                  sc_control *cntl)

int sysctl_errno;
char *sysctl_msg(int err)

int sc_ReadMsg(sc_sock *sd)
int sc_WriteMsg(sc_sock *sd)
int sc_CloseSocket(sc_sock *sd)

typedef struct {
     int flags;
     int port;
     int conn_timeout;
     int timeout;
} sc_control;

typedef struct {
     int status;
     char *result;
     sc_sock *sd;
} sc_result;

typedef struct {
     int sockfd;
     u_short msgtype;
     u_long msglen;
     char *msg;
     u_short status;
} sc_sock;
```

The sysctl() Routine

The **sysctl()** routine is a C library interface for communicating with **sysctld** servers. The **host** parameter is a pointer to a string that contains the name of the host to contact. The **op** parameter is a pointer to a string that contains the operation to run on the remote server. The **op** string can contain any sysctl expression and can be anything from a single command with no arguments to an entire script. The **sysctl()** routine does not interpret the **op** parameter. The **cntl** parameter is a pointer to an **sc_control** structure.

Before sending the operation to the remote host, **sysctl()** attempts to obtain a Kerberos ticket for the service **rcmd.***hostname*, where *hostname* is the fully qualified hostname of **host**. If a ticket cannot be obtained, the operation is sent to the server without any authentication information.

The result of the operation is returned via an **sc_result** structure. This structure contains the exit status of the remote operation as well as any output generated.

The sc_control Structure

The **sc_control** structure is used by the caller to control the communication mode between itself and the remote server. The **flags** field contains a bit-OR'ed set of flags, as defined in **sysctl.h**:

- **SC_NOWAIT**. The caller is not interested in the results of the remote operation. The server sends an immediate acknowledgement back to the caller and discards the result of the operation. Note that the acknowledgement is sent back after the authentication checks so any errors related to decoding Kerberos tickets are relayed to the client.
- **SC_NOAUTH**. No authentication information should be sent to the remote server.
- **SC_SOCKET**. The server is to return the result to the client through a TCP socket, rather than in the RPC return structure. This is useful in cases where large amounts of data are to be returned[3], when binary data is returned, or when the client wishes to send data to the remotely executing process.
- **SC_INTERACTIVE**. This is similar to **SC_SOCKET** mode in that a TCP communication stream is established between client and server. However, in interactive mode the client is connected directly to the remote interpreter rather than to an executing command. The **op** parameter is ignored in this case.
- **SC_PRIVATE**. Encrypt all communications between the client and server.
- **SC_NULL**. This flag causes **sysctl()** to make an RPC call to the **NULL** procedure on the remote server. The **op** parameter is ignored in this case. This provides a quick mechanism for determining if a **sysctld** server is up and running.

The **conn_timeout** field determines the length of time **sysctl()** waits for the initial connection to the server to be established. The **timeout** field is used to specify the maximum total timeout for the operation. If this maximum timeout is reached, **sysctl()** closes the connection with the remote server (which causes a **SIGHUP** signal to be sent to all remote processes) and returns an error. These timeouts only affect the RPC communication with the server and have no impact on the socket transfers if the caller specified **SC_SOCKET**. In all cases, the value of the timeout variables are interpreted as seconds. If they are set to zero, default values are selected.

The **port** parameter specifies the TCP port to use to connect to the remote server. If the port number passed is 0, **sysctl()** uses the *getservbyname* (3) system call to get the port number.

If **sysctl()** receives a **NULL** value for the **sc_control** parameter, it assigns default values for all fields. These default values are: **flags** = 0, **conn_timeout** = 10, and **timeout** = 1800.

[3]The maximum size of the result that can be sent back via the RPC structure is currently set at 1 MB.

The sc_result Structure

The **sc_result** structure is used to return information about the remote operation to the caller. The **status** field contains the exit status of the remote operation. The **result** field is a pointer to a string that contains the output of the remote operation, or a **NULL** value if the **SC_SOCKET** or **SC_INTERACTIVE** flags were specified. In the case of **SC_SOCKET** and **SC_INTERACTIVE**, the **sd** field contains a pointer to an **sc_sock** structure which is used to transfer data to and from the server.

Error Handling and Return Codes

If an error occurs during the transmission of a request to the server, **sysctl()** returns **NULL** and sets the global variable **sysctl_errno** to indicate the error. The **sysctl_msg()** library routine is used to return a pointer to a descriptive error message.

Otherwise, **sysctl()** returns a pointer to an **sc_result** structure. The exit status of the remote operation contained in the **status** field is determined as follows:

- If the remote server was unable to execute any portion of the operation, perhaps due to a syntax error or insufficient authorization, a value of **SCERR_TCLERROR** is returned in **status**. The **result** field contains any output produced up to the point the error occurred with the last line containing an error message describing the problem.
- If the remote operation was terminated by an **exit** statement, **status** contains the value of the parameter passed to **exit**.
- The shell exit status of the last external command executed, otherwise a value of 0.

The sc_sock Structure

The **sc_sock** structure is used in **SC_SOCKET** and **SC_INTERACTIVE** modes to transfer data between client and server. The **sc_WriteMsg()** and **sc_ReadMsg()** routines implement a simple message passing scheme that allows data to be sent to and received from the remote server. The use of message passing gives the client the ability to send and receive binary data, demultiplex **stdout** and **stderr** output from the remote process, and retrieve the exit status of the remote operation.

The **sc_ReadMsg()** routine is used to read messages from the server. It is passed the **sc_sock** pointer returned in the original **sysctl()** call. If an error occurs while reading from the socket, a value of -1 is returned and **sysctl_errno** is set to indicate the error. Otherwise, the **msgtype** field indicates the type of message received. The valid message types are:

- **SC_MSGSTD**. The **msg** field points to a static buffer of **msglen** bytes of **stdout** from the server.
- **SC_MSGERR**. The **msg** field points to a static buffer of **msglen** bytes of **stderr** from the server.
- **SC_MSGEOF**. The remote operation has terminated. The **status** field contains the exit status of the remote operation. The status field is interpreted in the same way as the exit status returned in the RPC structure.

The **sc_WriteMsg()** routine is used to send data to the remote server. The client must fill in the **msgtype**, **msg**, and **msglen** fields of the **sc_sock** structure. **msg** should point to a buffer containing **msglen** bytes of data to send to the server. If the message type is **SC_MSGSTD** or **SC_MSGERR**, the data sent appears as **stdin** to the remote process. If the message type is **SC_MSGEOF**, the server closes **stdin** of the remote process. The **msg** and **msglen** fields are ignored when sending an **SC_MSGEOF** message. If an error occurs while writing to the socket, **sc_WriteMsg()** returns -1 and sets **sysctl_errno** to indicate the error.

Note that if the **SC_PRIVATE** flag was set in the original **sysctl()** call, all messages to and from the server are encrypted. The encrypting and decrypting of the message data is handled within the **sc_ReadMsg()** and **sc_WriteMsg()** routines.

Appendix B – Code Samples

```
$ cat whoami.c
/*
 * This code sample takes the first argument to be the host to contact.
 * It uses the command "whoami" to echo the authenticated identity of the
 * user. The communication between the client and server will be encrypted.
 */
#include <sysctl.h>

#define CMD "echo Server $SCLHOST says I am [whoami]"

main(int argc, char *argv[])
{
     sc_control cntl;
     sc_result *r;

     bzero(&cntl, sizeof(sc_control));
     cntl.flags |= SC_PRIVATE;

     r = sysctl(argv[1], CMD, &cntl);
     if (r == NULL)
          printf("sysctl error: %s\n", sysctl_msg(sysctl_errno));
     else
          printf("%s", r->result);
}
$ cc -o whoami whoami.c -lsysctl -lkrb -ldes
$ whoami harlem
Server harlem.watson.ibm.com says I am vatore.@WATSON.IBM.COM
$
```

```
$ cat fscheck
#!/usr/agora/bin/sysctl -Lr
#
# Check filesystem usage - high-water pct mark is passed as a parameter
#
foreach fs [listfs] {
        statfs $fs sbuf
        set pct [expr $sbuf(used).0/$sbuf(size).0*100]
        if {$pct > $argv(1)} {
                set pct [format "%.2f" $pct]
                echo "$fs ==> ${pct}% Used, $sbuf(free) KB Free"
        }
}
$ cat /tmp/hostlist
harlem
badger
$ time fscheck -c /tmp/hostlist 80
harlem::/ ==> 84.29% Used, 1538 KB Free
badger::/usr ==> 98.35% Used, 3656 KB Free

real    0m0.29s
user    0m0.05s
sys     0m0.06s
$
```

Appendix C – Sysctl Command Usage Information

Usage: **sysctl** [*options*] [*command ...*]

Options:

-c *cluster*
: Run the command on the specified cluster. If the *cluster* argument contains a "/", it is assumed to be a file that contains the names of hosts in the cluster, one per line. Otherwise, the list of hosts in the cluster is retrieved by a call to Hesiod. More than one cluster can be specified with multiple -c options.

-f *num*
: Controls maximum fan-out. By default, a maximum of four concurrent connections is used. This allows greater (or lesser) parallelism.

-h *host*
: The server to execute the command on. More than one host can be specified using multiple -h options. If no -h (or -c) option is specified the server is assumed to be the local host.

-i *instance*
: Authenticate as this instance. The user is prompted for a password.

-L
: Provides an alternate way of delimiting output from multiple servers. Prepends each line of output from a host with the hostname, such as **foo.watson.ibm.com::...**.

-l
: Authenticate as the local server. The program uses the sysctl server key stored in the local Kerberos keyfile (**/etc/srvtab**) to obtain a ticket. This authenticates the user to the sysctl server as **rcmd**.*hostname*, where *hostname* is the local hostname. This is used to allow the local **root** user to gain access to sysctl commands reserved for authenticated users if the command is invoked by programs such as **cron**. The user must be running as **root**.

-n
: Send no authentication information.

-P *port*
: Specify the port number to use to connect to the remote server.

-p
: Use private (encrypted) communication.

-q
: Quick mode - Don't wait for the result from the server.

-r *file*
: Replay (drop) this file on the server(s).

-s
: The server should send the results back via a TCP socket. The output from the server is de-multiplexed into **stdout** and **stderr**.

-t *sec*
: Specify the connection timeout.

-T *sec*
: Specify the (RPC) remote operation timeout.

-u *user*
: Authenticate as this user. The user is prompted for a password.

-x
: Send a NULL RPC to the server(s) (like a ping).

command ...
: The command(s) to pass to the server.

If no command is given **sysctl** runs in interactive mode. The results of the server execution are printed to **stdout**. In the case of multiple servers, the output is delimited on a server-by-server basis for easy parsing.

Appendix D – Sysctld Command Usage Information

Usage: **sysctld** [*options*]

Options:

-A
Enable default authorization of ***.admin** principals.

-a *file*
Specify the name of the server ACL file (default: **/etc/sysctl.acl**).

-d
Run in debug mode. This causes more information to be printed to the log file and is useful for debugging problems with the server.

-k *file*
Specify the name of the file that contains the Kerberos server key (default: **/etc/srvtab**).

-l *file*
Specify the name of the log file (default: **/usr/adm/sysctl.log**). If the file specified is "syslog", the server logs all messages to **syslogd**.

-n
Run the server with authorization turned off. This should only be used for testing.

-P *port*
Specify the port number at which to register the service.

Any options given on the command line override the contents of the server configuration file.