A Reliable and Secure UNIX Connection Service

Dennis Draheim
Barton Miller
Steven Snyder

Computer Sciences Department
University of Wisconsin
1210 W. Dayton Street
Madison, Wisconsin 53706

Abstract

Distributed programs require a method for processes residing on different machines to identify each other and establish communication. One method is to provide a special connection service to perform this task. A good connection service should be easy to use. It should allow arbitrary processes to connect to each other as well as helping client processes to connect to server processes. It should provide location transparency; that is, the programmer should not have to know the network address of a process to connect to it. The connection service should be reliable. It should provide a way for a process to establish the identity of the user associated with the process to which it has connected, and to communicate securely with that process.

We have implemented a connection service for Berkeley UNIX that is reliable, available, secure, and easy to use. The connection service achieves ease of use through a simple interface based on the library routine **meet**. **Meet** allows one process to connect to another by specifying arbitrary names for itself and the other process. The connection service imposes no naming conventions of its own so it can be used with most name spaces and naming services. The service is location-transparent. It also provides a routine for posting services.

Reliable and available service is provided by replicating connection servers. Each server knows about all pending connection requests. The connection service provides continuous service as long as at least one server is running. Connections can be authenticated by an authentication server that works in cooperation with the connection server. Secure communication is achieved via the RSA public-key encryption algorithm.

The connection server was put in regular use in June 1986. Our limited experience indicates that it satisfies an important need of UNIX users.

1. Introduction

Distributed programs require a method for processes residing on different machines to identify each other and establish communication. Programmers should not have to worry about the details of establishing communication; they should be free to concentrate on the more interesting problem of coordinating a set of distributed processes.

The work of establishing communication between processes can be delegated to a special connection service. A good connection service should have several features. It should be easy to use. Its purpose is to ease the task of connecting processes to each other. It should allow arbitrary processes to connect to each other, as well as helping

Research supported by the National Science Foundation grant MCS-8105904 and a Digital Equipment Corporation External Research Grant.

client processes to connect to server processes in the system. The service should provide location transparency; the programmer should not have to know the network address of a process or the details of the network protocol to form a connection. The service should be reliable. It should not depend on the availability of any one machine. Ideally, the service should be usable as long as a network path exists between the two processes that wish to connect to each other. Optionally, a connection server should provide a way for a process to establish (beyond a reasonable doubt) the identity of the user associated with the process to which it has connected, and to communicate securely with that process.

We have implemented a connection service, called the *switchboard*, for Berkeley UNIX that is reliable, available, secure, and easy to use. The switchboard is implemented by a collections of server daemons and associated user library routines. A switchboard server accepts connection requests from client processes and returns the information necessary to create a interprocess communication connection between those processes.

We have also implemented an authentication server that verifies the identity of the user to whom a connection has been made, and a public-key encryption algorithm that may be used for secure communication once a connection has been established. Authentication and security are optional in our system. We do not force all users to communicate in a secure fashion; users choose between security and high performance.

The switchboard was designed for a network of heterogeneous machines running Berkeley 4.3 BSD UNIX [1]. We assume the existence of trusted hosts in the network, but not all hosts are trusted. A trusted host in one in which the users do not have physical access. We do not assume that the network itself can be trusted.

There are several issues that we are not addressing. We do not provide a naming service. Our mechanism provides a flat name space that allows processes to specify arbitrary names for each other. These process names must be unique within the environment (although if authentication is used, cases of non-unique process names will be detected). The names only exist in the server (and need be unique) while a connection is being formed. Our authentication and security mechanism does not address the detection of compromised keys, nor do we provide a means for secure key storage. Secure key storage is an administrative and physical security problem. We do not provide for authenticated one-way communication, such as in a mail delivery system, where the recipient of a message or agent process may not be on-line when the message is delivered. We assume both processes participating in a communication are on-line at the same time.

Several operating systems have connection services. In the DEMOS operating system [2], the connection service, or *switchboard*, accepts a request for connection to a process and returns a one-way communication channel that points to the desired process. The DEMOS switchboard is used mainly by operating system processes at system configuration time and has no reliability or authentication features. The Charlotte operating system [3] features a switchboard that acts as a connector for client/server connections only, but otherwise is similar to the DEMOS switchboard. Matchmaker [4], a connection server for 8th Edition UNIX, provides a network-independent connection service based on streams. Matchmaker allows processes to establish connections using the standard UNIX file I/O primitives.

Our switchboard design was influenced by name servers that were designed to provide reliable service. Grapevine [5] provides a replicated naming, authentication, and mail service for a distributed environment. The Clearinghouse system[6] binds names to network locations of objects. It also uses replication to provide reliability. The goal of both of these systems is to provide continuous service in the face of any single server failure. The switchboard's goal is to provide continuous service as long as there is at least one functioning server.

Related work in authentication and security includes the secure NFS system [7] developed by SUN Microsystems, which uses both public keys and DES encryption to achieve security. The SUN system is limited to authentication of login requests, and does not deal with secure interprocess communication. The switchboard's security and authentication mechanism is built around the RSA public-key encryption algorithm, and authenticates individual connections between processes, rather than login sessions.

2. Switchboard Description

We first describe the reliability features of the switchboard beginning with the user library-routine interface, then describe the implementation of the switchboard service. The second part of this section describes security issues and the authentication server.

2.1. Reliability Mechanism

The user sees the switchboard as a collection of library routines. These are:

meet(me, you)

Establishes a connection between a process calling itself *me* and one calling itself *you.* Returns the socket

for the connection to the other process.

putserver(me)

Registers a server process with the switchboard. A server process is willing to accept connections with anyone.

meetclient()

Used after **putserver**, establishes a connection between the server and a client process that has executed **meet**(me, server).

2.1.1. The Library Routines

The switchboard achieves ease of use through a simple interface based on the library routine **meet**. **Meet** allows one process to connect to another by specifying names for itself and the other process. For example,

Process 1	Process 2
sock = **meet**("me", "you");	sock = **meet**("you", "me");

Two processes need only agree on names for each other; they need not know each other's address. The connection setup proceeds as follows: Process *me* issues the call **meet**("me", "you"). The switchboard logs this request, and begins waiting. Then, process *you* issues the call **meet**("you", "me"). The switchboard compares this request with its list of pending requests, finds a match, and returns one end of the connection to process *me* and the other end to process *you*. The processes may then communicate with the UNIX routines **send** and **recv**.

The switchboard also provides support for forming connections to server processes. The **putserver** call creates a semi-permanent entry in the switchboard's list of pending requests. This entry will match a request from any other process. For example, after a server process has executed **putserver**("server"), a client process can connect to the server by executing **meet**("me", "server"). The server process calls **meetclient** whenever it detects that a client process is waiting to connect to it.

The **meet** routine works as follows. When the **meet**("me", "you") library routine is called, it first creates an interprocess communication socket and binds a network address to it. Next, it locates an operational switchboard server and establishes a connection with it. The meet routine sends a ‘‘meet’’ request, including the two process names, to the server. The server will eventually respond with the network address of a process that has executed the

corresponding **meet**("you", "me"). **Meet** then connects to the socket (process) whose network address it was given. If the switchboard server crashes at any time during the request, **meet** will detect this, locate another server, and resubmit the request.

Meet locates a switchboard server by searching a local file that contains a list of candidate hosts, as well as a pointer to the last host on which a server was successfully located. **Meet** first attempts to connect to a server by following this pointer. If this attempt fails or if the pointer does not exist, **meet** scans through the list of candidate hosts, attempting to connect to a server. This file is expanded dynamically as new servers join the system. This scan continues until a connection is established or the list is exhausted, in which case an error code is returned to the calling process. This *serial broadcast* method is used because UNIX does not provide a suitable broadcast mechanism. Processes need not agree on the location of a switchboard server with this mechanism. The **meet** routine forms a TCP/IP connection between processes. We also support a second version of **meet**, called **meet-d**, that supports datagram communication between processes.

2.1.2. Switchboard Server Internals

The switchboard employs multiple servers running on different machines to increase the availability of the service. These servers are arranged in a fully connected network; each server is connected to all other servers. Whenever a server receives a request, it propagates it to all the other servers. This is necessary because two clients who wish to connect to each other may send their requests to different servers. Using this mechanism, the switchboard provides continuous service as long as at least one server remains operational.

Each server uses the following protocol when accepting meet requests from client processes:

1. Wait for meet request.
2. Accept request from process *me*.
3. Save request in table for later use.
4. Propagate request to all other servers.

In the simple case, where both client processes connect to the same server, the protocol in Figure 1 is used. Note that connections are asymmetric. The switchboard server responds differently to the two clients. The client designated active will take the active role by connecting to the network address of the other client. The passive side will accept the connection from the active side. This asymmetry becomes more important in the multiple-server case. The meet request must be removed from each server's table during the cleanup phase. The server making the connection sends a message to all other servers telling them to do this.

In a switchboard with multiple servers, it is possible for two client processes to connect to different servers when making their **meet** requests. The servers use the protocol in Figure 2 to establish the connection in this case. If a server should fail during this protocol, the responses are not sent to the clients. A client will detect the failure on the termination of its connection to the server and will connect to another server to resubmit the request.

A race condition may occur in our protocol when client *me* sends the request **meet**("me", "you") to the first server at the same time that client *you* sends the request **meet**("you", "me") to the second server. Both servers will save the request in their table and propagate the request. Each server must perform an arbitration when it detects the race condition. This insures that one client is sent the response ‘‘matched-active’’ and the other is sent the request ‘‘matched-passive"’’ as required by the protocol. Each server attaches its host identification number to each message it sends. On receiving a message, the host with the greater host id will send the ‘‘matched-passive’’ message to the other server. The host with the lesser host id will do nothing. The race condition translates to the Multiple Server Protocol in Figure 2.

When a server fails, all other servers detect this failure by the loss of the connection to the failed server, update their list of operational servers, and remove all table entries that originated from the failed server. The client will also detect the failure and resubmit its request to another server. When a server recovers from a crash or executes for the first time, it integrates itself into the server network using the protocol in Figure 3. The recovering server uses the same location algorithm as **meet** to find a running server.

2.2. Authentication and Security Mechanism

The user library provides the following routines for establishing secure connections.

securemeet(me, you, myusername, yourusername)

Establishes a secure connection between *me* and *you.* Returns a socket for the connection to the other process and *youruniquekey*, the public key for process *you.*

encrypt(cleartext, clearlength, key, cyphertext, len)

Encrypts the cleartext message using *key.*

decrypt(cyphertext, cypherlength, key, cleartext, len)

Decrypts the cyphertext message using *key.*

If a pair of processes desire an authenticated connection, they call **securemeet** instead of **meet**. The extra parameters *myusername* and *yourusername* are unique user names that must have been previously registered with the switchboard's authentication service. When **securemeet** returns successfully, process *me*, which must be owned by user *myusername*, is guaranteed to be connected to a process calling itself ''you'', which is owned by user *yourusername*. This guarantees that no other user with a process called ''you'' can connect to process *me*, either inadvertently or on purpose. Once connected, the processes may use the routines **encrypt** and **decrypt** to encrypt and decrypt the data with the keys returned by **securemeet**.

The authentication mechanism of the switchboard comprises two entities: the authentication server and the authentication administrator. The authentication server is a process whose main job is to maintain a key directory; the administrator is a person responsible for authentication of initial keys.

2.2.1. Authentication Server

The switchboard's authentication service is provided by a server running on a trusted host. The authentication server maintains a public key directory and regulates access to it. Each entry in the directory consists of a unique name and public key. The unique name is chosen by the user and is checked for uniqueness by the authentication server before it is entered in the directory. The user is free to choose any unique name, although the system administrator may impose a standard on the assignment of names. The authentication server does not enforce any restrictions on the names used.

Several library routines are used to communicate with the authentication server. These are:

insertkey(username, publickey, requestername)

Inserts key *publickey* and its associated *username* into the public key directory. *Requestername* is the originator of the request.

getkey(username, publickey, requestername)

Returns the key *publickey* associated with the directory entry for *username.*

changekey(username, newpublickey, requestername, oldsecretkey)

Changes the key associated with the entry for *username* to the new key *newpublickey.* Uses the key *oldsecretkey* to authenticate the requester.

deletekey(username, requestername)

Deletes the entry for *username* from the public key directory. Uses the key *secretkey* to authenticate the requester.

markOK(username, adminsecretkey)

Marks the entry associated with *username* as suitable for distribution. Uses the authentication administrator's secret key to authenticate. This call can only be executed on a trusted host by the system administrator.

initauth(adminpublickey, adminsecretkey)

Starts up the authentication server daemon for the first time and initializes its keys. This call can only be executed on a trusted host by the system administrator.

Securemeet calls the routine **getkey** to find the public keys for the users making a connection. It then authenticates the users using the authentication protocol described below. Users call the routine **insertkey** to create a new entry in the public key directory (subject to the authentication administrator's approval). Likewise, users call **changekey** and **deletekey** to change and delete key entries, respectively. The authentication server uses an authentication protocol to ensure that only the owner of an entry may change or delete it.

2.2.2. Authentication Administrator

The authentication administrator is a person responsible for maintaining the integrity of the unique name database. When a user submits a {unique name, public key} pair using **insertkey**, the new entry in the public key directory is marked ‘‘tentative’’ until the authentication administrator has verified that the user name is valid. This must be done through some secure, out-of-band means, for example, via private phone line or trusted courier. The means of initially authenticating a user name will depend on the particular needs of the users. After the name has been authenticated, the authentication administrator marks the entry "ok for distribution" with the **markok** library routine (which may only be called by the authentication administrator). At this point, the authentication server will begin accepting requests for the public key associated with this entry. Inserting a key is a relative infrequent event, typically corresponding to the addition of a new user to the system. If the name is not authenticated, the administrator can delete the entry.

2.2.3. Encryption Algorithm

The switchboard's encryption method is an implementation of the public-key encryption algorithm, RSA [8]. We chose RSA because it has (so far) proven itself safe and can easily be extended to compensate for improving technology by increasing the size of the keys. Our current implementation uses 100 decimal digit keys, which are large enough to render attempts to crack the system computationally infeasible (requiring several days of CPU time on a Cray 1 computer). Additional security may be gained by changing keys periodically; the length of the period determines the level of security.

2.2.4. Authentication Protocol

The entire connection establishment protocol involves messages between the user process (executing the library routine **securemeet**) and the switchboard daemon, between the user process and the authentication server, and between the user process and the process to which it wants to connect. This protocol is based on the public-key protocol presented in [9], and [10] with the addition of the switchboard connection messages. Figure 4 summarizes the protocol.

2.2.5. Initial Key Distribution

Any security mechanism that depends on keys has the problem of how to distribute the initial keys. In the switchboard, the problem is limited to that of distributing the authentication server's public key. This must be done out-of-band through some secure but not necessarily secret means, for instance, by reliable courier.

3. Evaluation

How does the performance of the switchboard system compare with the best performance achievable under UNIX? We believe it is quite favorable. In the simplest case, if there is a single switchboard server, a connection request will require four messages to be exchanged. The minimum number of messages that must be exchanged to establish a UNIX connection is two. This assumes that one of the processes knows the network address of the other, so the two extra messages are a small price to pay for location transparency.

3.1. Reliability and Performance

We can calculate the performance penalty of adding reliability to the switchboard system by comparing the number of messages exchanged for a replicated server to the number required for a single server. In the single server case, four messages are required to establish a connection. Replication adds two messages for the server handshake and $2(N-1)$ messages for the propagation and subsequent cleanup of meet requests, where N is the number of servers. We see that the increase is linear in the number of servers and since N will normally be small, the increase in message traffic will not be significant.

3.2. Authentication, Security, and Performance

The addition of authentication has some affect on performance. An authenticated connection request results in a total of 15 messages being exchanged (4 for the **meet**, 4 for connections to the authentication server, and 7 for the authentication protocol). Our benchmarks show that a connection request (using 100 decimal digit keys) requires about 4 seconds on VAX 11/750's. We do not feel this is an outrageous price to pay for authentication. Most of this time is due to the implementation of the RSA encryption algorithm in software; with the addition of hardware assistance for the RSA calculations, the times could be reduced. An addition that would help performance is the local caching of keys. The current implementation requires each connection request to go through the entire authentication protocol. If, instead, we allowed users to store keys locally once they were obtained from the authentication server, subsequent connections to the same user would require only three of the seven authentication protocol messages, thus reducing the total number of messages to eleven.

4. Testing and Current Status

The switchboard has been implemented and is running on a collection of VAX 11/750, VAX 11/780, and MicroVax-2 computers connected by a 10 megabit Ethernet. The current version has been tested for robustness and fault tolerance. A state diagram of client/server and server/server interactions was constructed and the server was killed at every point before and after each type of message was sent and received. The server proved reliable in all cases tested.

Testing the security of the system is more difficult and is ongoing. Our previous testing has been informal and a complete study of security is being done now. We believe that most security breaches involve some form of carelessness in administration, rather than inherent problems with the security algorithm. Careful administration is

required to prevent these problems.

Our current tests involved a precise definition of the threat model and a state transition model of the servers' behavior. We are enumerating each state in the severs' protocol and verifying that each steps operates correctly given all inputs. When this analysis is completed, we will verify that the code that implements the servers matches the protocol specification. Results from this study will appear in a future paper.

The switchboard service is currently being used by several projects, including UPCONN, a graphic layout tool for distributed programs [11]. The switchboard simplified UPCONN's task of establishing the connections needed to execute the distributed program.

5. Future Work

The switchboard has been in use since June 1986. Our experience with the switchboard has indicated the need for some additions and changes. First, although the switchboard daemons are replicated, the authentication server is not. While a replicated authentication service would be more reliable, such replication causes several serious security problems concerned with maintaining the integrity of the public key directory, which would also need to be replicated. Second, it would be more efficient, although possibly less secure, to allow users to cache keys locally once they had been obtained from the authentication server. The user would have to check periodically with the authentication server to see whether the key had changed, otherwise there would be the danger of using an obsolete and possibly compromised key. Third, the switchboard's server location routine becomes inefficient as the number of daemons increases. It would be useful to have a broadcast mechanism for locating switchboard daemons. Fourth, we envision the incorporation of a naming service on top of the switchboard. A mechanism providing a hierarchical name space could greatly ease the unique name requirements in the system. Last, the RSA encryption algorithm software implementation is too slow for many applications. We are currently designing hardware to accelerate the computations needed to implement RSA.

Acknowledgements The authors would like to thank Paul Onnen and Stephen Stukenborg for their work on the switchboard replication mechanisms.

6. References

[1] W. Joy, E. Cooper, R. Fabry, S. Leffler, K. McKusick, and D. Mosher, ‘‘4.2BSD System Manual,’’ Computer Systems Research Group Technical Report, University of California, Berkeley (July 1983).

[2] Forest Baskett, John H. Howard, and John T. Montague, ‘‘Task Communication in DEMOS,’’ *Proceedings of the 6th SOSP, Operating Systems Review* **11**(5) pp. 23-31 (November 1977).

[3] Yeshayahu Artsy, Hun-Yang Chang, and Raphael Finkel, ‘‘Charlotte: Design and Implementation of a Distributed Kernel,’’ Computer Science Technical Report #554, University of Wisconsin, Madison, Wisconsin (August 1984).

[4] David L. Presotto, ‘‘Matchmaker: The Eighth Edition Unix Connection Service,’’ *Proceedings of the 1986 European Unix Users Group Conference*, Florence, Italy, (April 1986).

[5] Andrew D. Birrell, Roy Levin, Roger M. Needham, and Michael D. Schroeder, ‘‘Grapevine: An Exercise in Distributed Computing,’’ *Communications of the ACM* **25**(5) pp. 260-274 (April 1982).

[6] Darek C. Oppen and Yogen K. Dalel, ‘‘The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment,’’ XEROX Technical Report OPD-T8103 (October 1971).

[7] Bradley Taylor and David Goldberg, ‘‘Secure Networking in the Sun Environment,’’ *Proceedings of the Summer Usenix Conference*, pp. 28-37 Atlanta, Georgia, (June 1986).

[8] R. L. Rivest, A. Shamir, and L. Adleman, ‘‘A Method for Obtaining Digital Signatures and Public-Key Cryptosystems,’’ *Communications of the ACM* **21**(2) pp. 120-126 (February 1978).

[9] Roger M. Needham and Michael D. Schroeder, ‘‘Using Encryption for Authentication in Large Networks of Computers,’’ *Communications of the ACM* **21**(12) pp. 993-999 (December 1978).

[10] Gerald J. Popek and Charles S. Kline, ‘‘Encryption and Secure Computer Networks,’’ *Computing Surveys* **11**(4) pp. 331-356 (December 1979).

[11] Mitali Bhattacharyya, David Cohrs, and Barton Miller, ‘‘The Implementation of a Visual UNIX Process Connector,’’ Computer Sciences Technical Report, University of Wisconsin (August 1986).