A
Remote Procedure Call
Mechanism for Unix

Martin Guy

1984

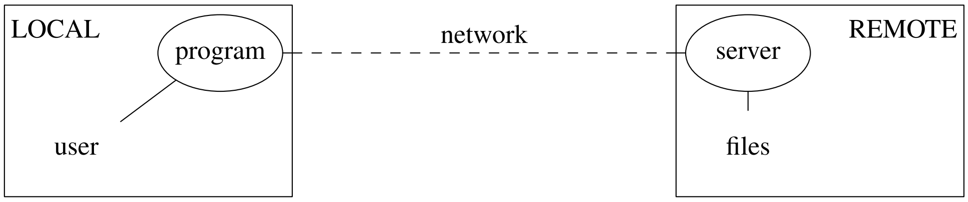
A Remote Procedure Call mechanism for UNIX

This report describes the design and implementation of a Remote Procedure Call mechanism for UNIX using Virtual Circuit Transport Service Byte streams. (TSB)

A remote procedure call is an extension to the concept of procedures used in third generation languages such as Pascal, BCPL and C. When it is called it effects a change on or returns a value dependent upon the state of a machine other than the one upon which the main program is running.

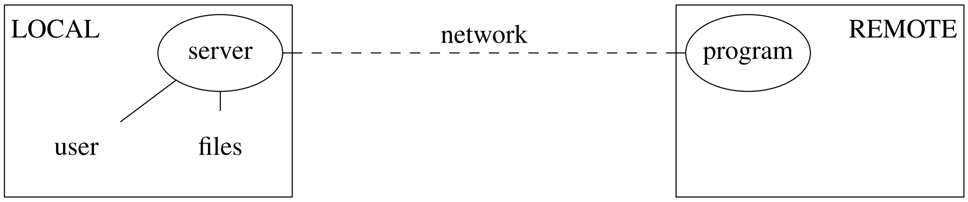
This has two applications; Remote File Systems where files on other machines can be manipulated as though they were resident on the program's own host and Remote Execution which allows a program to be run on one machine in the environment of another.

The Remote File System (RFS) is a means for accessing files on many machines as though they were resident on one. Some part of a filename identifies it as a local or remote file to the RFS software which sends a message to a server on the machine on which the file is in fact held instructing it to access the file and return the results.



Remote File System

Remote Execution (REX) takes this one stage further. By some means the program you want to execute is set running on another machine. All input and output is then performed by a server running on the user's machine under the control of the program. By this means, processor-intensive programs can be run on dedicated or less busy machines and a program which is only available on one host can be accessed from others. Good targets for this are programs which require much processing but not too much I/O eg. compilations, data compression, text formatting.



Remote Execution

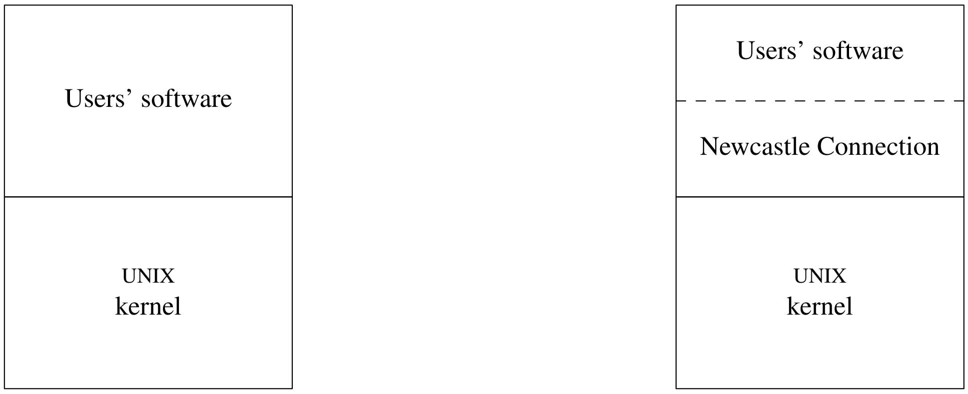
I wanted it to be possible to compile unmodified standard system programs with the remote procedure call software packages to produce a remotely executing version of the same program or one which can access remote files.

As UNIX provides all services such as file access by a set of procedures (''system calls'') it is convenient and general to implement RPC by providing a set of procedures which present the same interface to the calling program as the system calls but which communicate with servers to do the work.

A naming scheme is required for the Remote File System software to distinguish between local and remote file accesses. I arbitrarily chose "filename@host" which conforms to local file transfer protocol conventions. In UNIX a file is identified to system calls by its unique pathname and the '@' character is unlikely in a filename. If it were necessary to access a local file on host eagle whose name contained an '@' character it would merely be necessary to append "@eagle" to the filename.

Comparison with other RPC systems

The Newcastle Connection (Lindsay Marshall et al, University of Newcastle upon Tyne)



Perhaps the most well known distributed computing system, the Newcastle Connection is a software library which catches a program's system calls and determines whether to do local or remote accesses before calling the kernel. Remote machines are specified by a natural extension to UNIX filenames, a super-root directory one level above the root of each machine's filestore containing the names of other accessible hosts. An example of a remote filename is ''/../eagle/usr/fred/work''.

It supports changing one's working directory to a directory on another machine and remote execution is specified simply by typing the name of an executable file on another machine. (eg. typing ''/../host2/bin/ls /usr/fred'' on host1 would list the contents of directory ''fred'' on host1 but the ls program would run on host2.)

The Newcastle Connection provides both remote execution and remote file access with one natural and general interface. For it to be fully general, all programs should be compiled with the Newcastle Connection software.

UNIX supports the sharing of the text segment in memory of several invocations of the same program, but does not share the text of software libraries common to different programs. As all programs contain the object code of the Newcastle Connection, this would be a considerable burden to a memory-bound machine.

Each program also contains the code to implement a virtual circuit transport service; at UKC this is implemented in the kernel, reducing memory usage and providing a faster transport service than could be achieved by a user-space implementation.

Note that a kernel implementation of the Newcastle Connection is almost complete at the time of writing, but I have not had an opportunity to use it yet.

ADHOC RPC mechanism (Peter Collinson, University of Kent at Canterbury)

Implemented in a very short time, its aim was to provide a few simple RFS services with little programming effort. As with my mechanism and the Newcastle Connection, it provides a set of procedures to replace selected system calls.

You need to modify programs which are to use it; the remote version of ls, for example, takes an extra argument to specify the target host. It works and has done so for a year.

Implementation

Remote Execution has one great advantage over the Remote File System – it is easier to implement. I have written the Remote File System software but time has not permitted me to type it in and work on it.

The remotely executing program communicates only with its server. A Remote File System program also interacts with its own file system and could be in communication with several other machines. It therefore has to maintain a list of which open files are local, which are remote and with which host to communicate for each remote file.

Remote Execution is also a more clearly defined problem: that it must appear to the remotely executing program in every way that it is running on the machine on which, in fact, its server is running. No file descriptor mapping table is necessary; the main program can deal directly in the actual file descriptors used on the server's machine as it has none of its own (save that open to the transport service).

Security is also less of a problem. Any attempt by the remotely executing program to interact with the machine it is running on is trapped by the Remote Execution library and passed to the server on the home machine.

Program Description

1. Libraries

The virtual circuit transport service is implemented in the kernel at Kent. The interface to it is by special devices "/dev/tsb??". transport service commands are issued by *write* system calls and incoming packets are received by *read* system calls.

Convenient program interface to the transport service is provided by Peter Collinson's *libtsb* package. This contains functions such as *gettsb( )* which tries to find a free TSB port and return a file descriptor open to it, and *tsconnect(fd,called,calling,quality,explan)* which builds a command structure for the transport service, issues the CONNECT message to the port and returns a pointer to a structure containing the reply.

The file *trans.c* uses these to provide functions which treat the transport service as a pure byte stream and send and receive specific data objects. For example, *sendshort(s)* transmits a 16-bit integer to be received by *recvshort( )*. The −byte, −short and −long primitives deal with one-, two- and four-byte numbers. The −buf and −string primitives deal in buffers of data as used in *read* and *write* system calls. The −char and −string functions may perform character set mapping on the data whereas the corresponding −byte and −buf functions deal in bit patterns.

Trans.h contains appropriate type declarations for the routines in *trans.c* for inclusion by files which call them.

2. Remote Execution

The primitives in *trans.c* are used by the functions in the file **rpccalls.c**. These functions supplant UNIX system calls dealing with the process' environment. They send the arguments to the remote server and receive the return values back.

3. The server

Rpcsrv.c consists of one large case statement switching on command tokens. It receives parameters, makes a system call and sends the results back to the master program using the primitives from *trans.c*. There should be a close similarity between the functions in *rpccalls.c* and the corresponding cases in *rpcsrv.c*.

4. Remote File System

The remote file system is built on top of the top-level functions of the remote execution software. **Rfscalls.c** contains another set of fake system calls which determine whether a file access is local or remote and then either perform the real system call or the remote system call from *rpccalls.c*.

Implementing the Remote File System

Modifications required to the Remote Execution software

The main difference between the Remote File System and the Remote Execution software is that the Remote File System software must deal with the file system of the host upon which it is running and perhaps several other hosts.

This implies that

a) The file descriptor numbers which the user program deals in no longer correspond to the real file descriptors as was possible with Remote Execution. The software must maintain a table recording which open files are local and which are remote. on which hosts to map the user's file descriptors to the appropriate host and actual file descriptors. Care must be taken that user file descriptors are allocated in the same manner as the kernel, ie the lowest numbered free file descriptor is used first. Some programs rely upon this.

b) Transport primitives must take an extra argument, the file descriptor open to the host with which they are to communicate. Only one set of buffers is necessary for communication with several hosts because after each remote procedure call (which deals with only one host) the I/O buffers are always empty.

c) An extra set of imitation system calls is required which determine whether each call refers to a local or remote object and which call either the real system call or the remote one. The remote ones are used directly by REX. These too must take the appropriate file descriptor as an extra argument. I suggest that rpccalls.c be given a conditional compilation flag of the form

```
#ifdef REX
# define tsfd 0
#endif

#ifdef REX
open(path, flags, mode)
#else
rpc_open(tsfd, path, flags, mode)
#endif
char *path;
{       ...
```

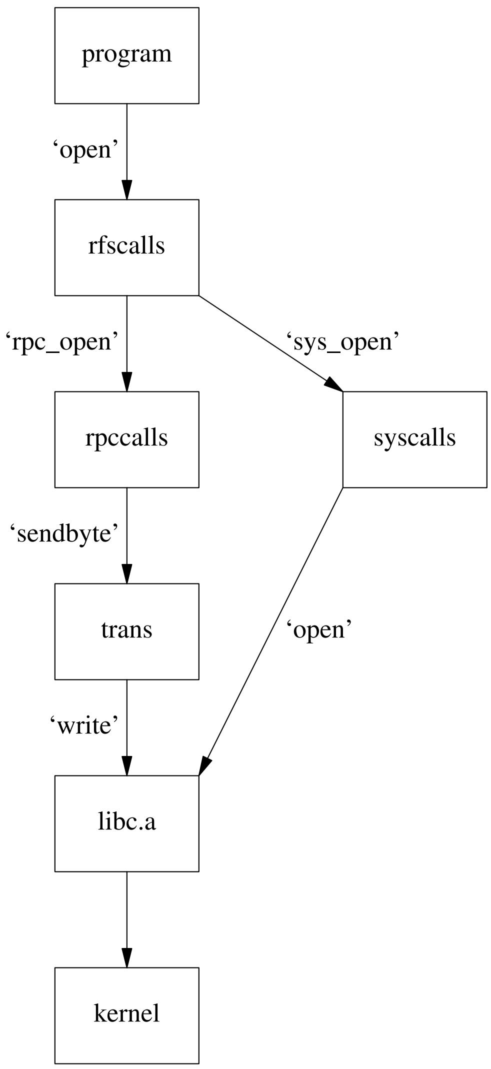
to avoid the introduction of (yet) another function called ‘‘open’’. (Consult *rpccalls.c* to see how this fits in)

d) Another file will be required to give the RFS system calls access to the real system calls. I suggest

```
. . .
int
sys_open(path,flags,mode)
char *path;
{
       return(open(path,flags,mode));
}
. . .
```

This is needed to get round the linking and loading as it is not possible for a function to call another function with the same name as itself. Selex should be used on this to hide the references to the real system calls in the object file.

e) the RFS software must be able to set up a transport service connection to any host at will as it cannot be known in advance which hosts it will be required to communicate with until a special filename is encountered.



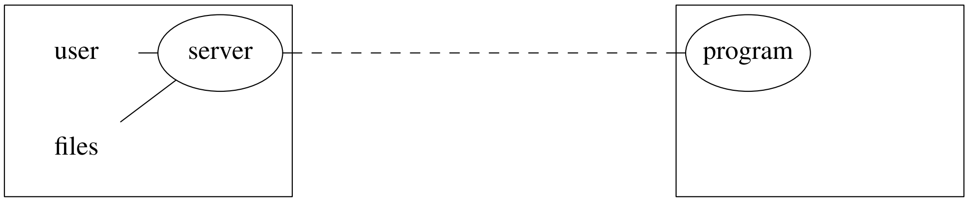
Calling Hierarchy of Remote File System

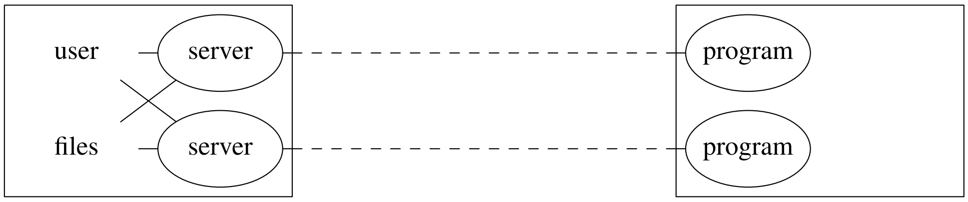
Most system calls are quite simple to implement; the program sends the arguments, the server does the work and returns the results. There are, however, several areas in which this approach is insufficient. For these it helps to remember that the remotely executing program should be unable to tell that it is not running on the home machine.

1) Fork

The *fork( )* system call causes creation of a new process (‘‘the child’’) identical to the parent in every way except that *fork* returns the process-id of the child to the parent and 0 to the child.

To emulate this, both the main program and its server must fork at the same time and a new transport service connection must be established for communication between the child program and its server.





The child server then returns 0 to its master and the parent server returns the process-id of the child server.

The *wait* system call which is associated with *fork* can be implemented as simply as most other calls – the server waits for the child server to exit.

2) Exec

The remotely executing program deals only in files on the local machine. As the new program to execute is referred to by a file name, it must run on the local machine. Open files remain open across an *exec*. As the server is in possession of the file descriptors open to real files, it is again appropriate that the server should execute the new program.

If the server fails to execute the new program (for example, because the specified file does not exist) it sends back the return value from exit to the master. Otherwise it dies in the *exec* and the transport service automatically issues a *disconnect* message to the master program. When the master’s *trans* primitives receive this instead of the data they were expecting, they exit directly; control is not returned to the main program.

For the Remote File System I have found no fully satisfactory way to *exec*. If remote files are to remain open across an *exec* (as they should) the new program must also be compiled with the RFS software and the table of local and remote file descriptors must somehow be passed to the new process. There is no perfect way for RFS programs to exec non-RFS program as the latter cannot deal with the open remote files.

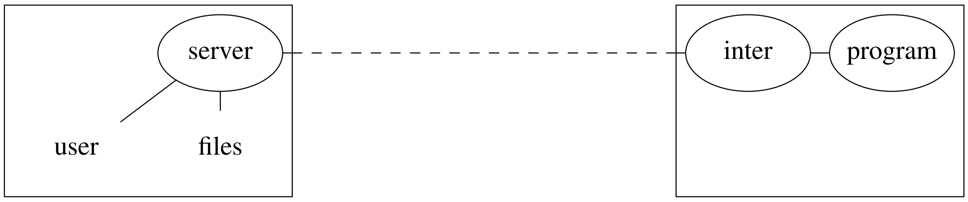
3) Pipes

It would be most efficient for the master program to create a pipe on the remote machine, which is still consistent with not interacting with its environment. However, this loses the generality of all its file descriptors corresponding to the real ones of the server. Also, because the usual actions following the opening of a pipe are fork and exec, it is desirable that the pipe is created on the local machine by the server. This makes it not a special case.

4) Signals

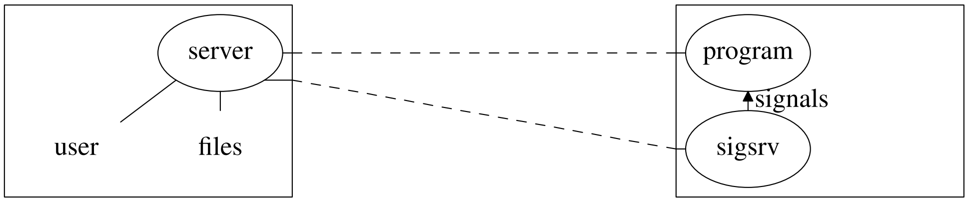
Process-ids are meaningful only on the local machine, so the sending of signals must occur there. If a server is sent a signal, it must transpire that the same signal be sent to its master; if it has trapped that signal it may wish to take action – a visual editor, for example, may wish to clear the screen and reset the terminal modes before stopping and to redraw the screen when it is restarted.

It is apparent that the transport primitives in the master program cannot generate the signal as signals are asynchronous and the transport primitives are called synchronously with the master's system calls. If the master were to enter the 'stopped' state due to a signal, it would not be listening to the transport channel and so would not read a message to restart. A second process is needed to deliver signals asynchronously to the master.



a) An intermediate program ''in series'' with the transport service connection to handle the transport service, pass data to and from the master via a pipe and to deliver signals to the master. This incurs costs in efficiency and the intermediary must have full knowledge about the data transmission format so that it can identify both requests for signals in the byte stream from the server and instructions to fork along with the server and master programs. Alternatively a packet structure could be imposed upon the byte stream containing a flag to say whether each packet is to be acted upon by the intermediary or forwarded. Either way, the intermediary will have to be ready to accept data from more than one source. Different UNIX systems have different mechanisms for doing this, but some may have none.

This approach would also greatly increase the complexity of the system.



b) A separate, asynchronous process to deliver signals to the master is a better solution, the main drawback being that it doubles the number of transport service ports required on both machines. A separate signal server is required for each remotely executing process. One universal signal server would alleviate this problem a little but incurs problems such as validation and deciding who is allowed to send signals to which processes. If each signal server runs with the same user-id as the master program, this validation is done by the kernel.

Linking and Loading

Great problems are encountered when trying to load the separate object code modules into the final program. In a remotely executing program there are, for example, two functions called *write*, the fake one and the real. The transport software must cause a real system call but calls from the user's program must call the fake ones from *rpccalls.c*. In a RFS program there are three!

Each of the –.c files is compiled to produce a corresponding –.o file containing, among other things, the object code for the –.c file and a list of functions which were called by functions in the file but which were not themselves included in the file. (''Unresolved references'')

The UNIX program *ld* is used to resolve such references between many object code modules to produce the final executable program. It also allows you to combine several object code modules into one which may still contain unresolved references to be resolved by further loading. Its behaviour is such that the following incantations successfully produce a remotely executable program from the separately compiled files.

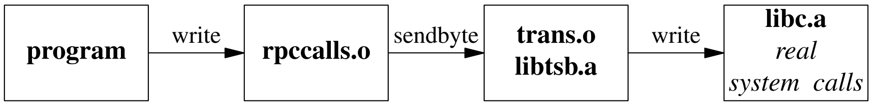
```
ld -r -o t#.o trans.o libtsb.a libc.a

ld -r -o rex.o rpccalls.o t#.o

ld -o program program rex.o libc.a
```

The –r flag to *ld* means that the object file produced may still contain unresolved references. The –o flag specifies that the next argument is the name of the file in which the result is to be put. Libc.a is a library of object code files containing the real system calls. Names beginning with ''t#'' are the UNIX convention for temporary files.

The calling hierarchy is as follows:



The first *ld* finds references to system calls in trans.o and libtsb.a and fills in the addresses of the real system calls from libc.a.

The second line burps when it finds a second occurrence of a function called ''write'' when it inspects *t#.o* as it has already found one in *rpccalls.o*; it throws the second occurrence away.

The third command finds that it can resolve references by *program.o* to ''*write*'' when it inspects *rex.o*; when it gets to *libc.a*, references to the replaced system calls have already been satisfied by *rex.o*. Only functions which are not replaced by *rex.o* are extracted from *libc.a*. Library functions which program.o requires from libc.a and which in turn require system calls are loaded in and linked with the fake system calls from *rex.o* which *ld* has already found.

Selex

There remains a problem which *ld* cannot be used to solve if *trans.o* or *libtsb.a* call and have been loaded with any library functions from *libc.a*. For example, *printf* produces prettily formatted output and then calls *write* to put it out. The first stage of loading will insert the code for *printf* into the object file *t#.o* with the references to *write* linked to the real system call as it should be.

When *program.o* comes to be loaded, its references to *printf* will be linked with the *printf* already in *rex.o* which calls the real system call '*write*'. What should have happened was that a second copy of *printf* should have been extracted from *libc.a* and its references to *write* linked to the fake system calls in *rpccalls.o*.

The best solution to this problem, a solution which also removes a potential bug since the behaviour of *ld* is not defined when there are multiple occurrences of the same function name, is a program to remove unwanted symbols from an object file.

Selex takes a compiled object file in '*a.out*' format and marks selected function names as 'local' to that module; they will not be used to resolve references from other files again. (It Selectively Exports functions from an object file.) This enables you to hide both system calls which are to be replaced and library functions which have been loaded with the real versions of replaced system calls.

Interestingly, after I had written *selex* a problem was encountered by Sean Leviseur porting a program to the orion super-microcomputers which only selex (or a revision of the system's libraries) could fix.

NAME

selex – selective exporter

SYNOPSIS

selex [-x] [-v] file.o _symbol ...

DESCRIPTION

Selex takes a file in a.out(5) format and marks selected symbols as 'local' so that they will not participate in further uses of *ld* . Symbols will normally start with an underscore character.

–x This option specifies that all symbols are to be made local *except* those in the argument list.

–v Verbose mode. The names of all symbols made local are printed on the standard output.

DIAGNOSTICS

If a symbol in the argument list does not occur in the object file, a warning is printed on the standard error output.

AUTHOR

Martin Guy, UKC

HISTORY

Written to make loading of Remote Procedure Call mechanism possible.

SEE ALSO

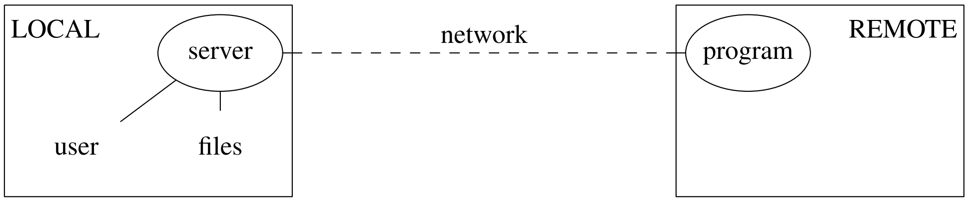
ld(1), a.out(5)

FEATURES

Selex –x should check that all symbols in the argument list occur in the –.o file before it zaps any of them as a guard against typos.

Setting up a remote execution

A remote execution requires the following situation to be set up



So that the same server program (“rpcsrv”) may be used for both REX and RFS applications, the server performs no validation checking and is not capable of setting up transport service connections. It is also convenient that the setup and running phases be divided into two separate programs.

The program initially called by the user is the same for all REX programs (“rpcinit”) but is known by many names. It establishes a connection to the program “rpcboot” on the host upon which the real program is to run.

Rpcinit sends to *rpcboot* its list of environment variables and the list of arguments it received to pass to the real program. *Rpcboot* then execs the required program with the transport service connection on file descriptor 0 and *rpcinit* execs *rpcsrv*, which takes one argument, the number of the file descriptor open to the transport service. It is then up to the program to continue the conversation by issuing its first trapped system call.

Security

The reliable denial of services to unauthorised requests

It is extremely important that the Remote File System server is given privileges appropriate to the user who is using it to access files on a remote machine.

Validation of requests for a remote execution is not as important. While the job is running resources may be being used illicitly, but no permanent harm can be done on the scale that an unauthorised remote file system server could wreak, because a remotely executing program does not interact with the file system of its host. The following discussion is concerned primarily with authorisation in the remote file system.

It is a first principle in maintaining security that access should not be determined by knowledge of information alone because information can be copied and used by others without the victim realising.

Unfortunately, the requirement of a physical element (such as being logged in to a terminal adjacent to the machine) is an unacceptable restriction on the utility of a remote access system. We must devise a means by which a request can reliably be accepted or refused based solely on the exchange of information.

In the UNIX environment, ''privileges appropriate to the user'' are conferred by having control of a process which has the effective user-id of the user in question. This can be achieved by logging in to the machine as the user, which requires knowledge of the user's username and corresponding password, each of which consists of up to eight characters. This is therefore sufficient information to allow access to a user's files.

The password for a user is held in an irreversibly encrypted form in a publicly readable UNIX file. The test for validity of a password is to encrypt the supplied password and to compare the result with the encrypted version held in the file. It is the knowledge of the unencrypted password which allows access to files, not that of the encrypted one.

Validation

A full explanation of the problems in secure validation will not be presented here.

The characteristics of the setup of a transport service connection are as follows:

A server on the remote machine opens a TSB port and **listens** for a specified **name** on it.

The client opens a TSB port on his machine and sends a connect message specifying the host and the **name** it is connecting to.

The listening server is then informed which host the incoming connection came from.

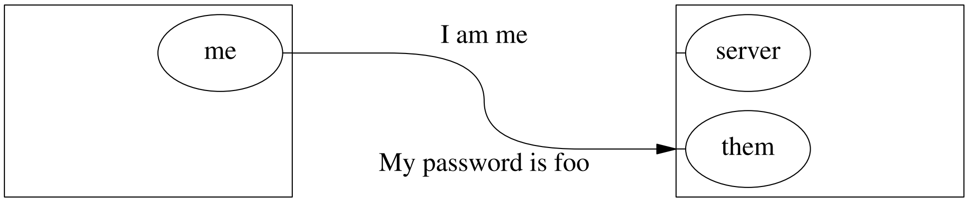
Once this has happened a bidirectional byte stream has been set up between the processes. It cannot be monitored of interfered with by an intruder because the open transport ports refuse a second attempt to open them.

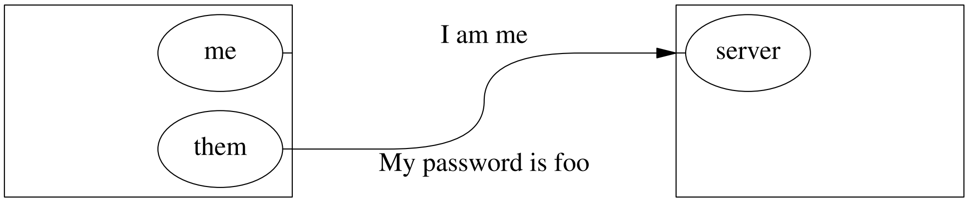
Two programs on the same host can listen for the same name on two ports. An incoming connection for that name is only passed to one of them; the one which listened first appears to be favoured.

Proposal

The connecter must supply the correct unencrypted password to the listener who can then encrypt it and check it against the system password file.

Unencrypted passwords must not, therefore be passed over the network. An intruder could listen for the same name as the server, discover the user's password and use the knowledge to fool the server:





Restricting access to TSB ports is not a satisfactory solution because the RFS software needs to be able to obtain TSB ports at random when it is running with an unprivileged user-id. It would also mean that each machine is dependent upon the security of all the others.

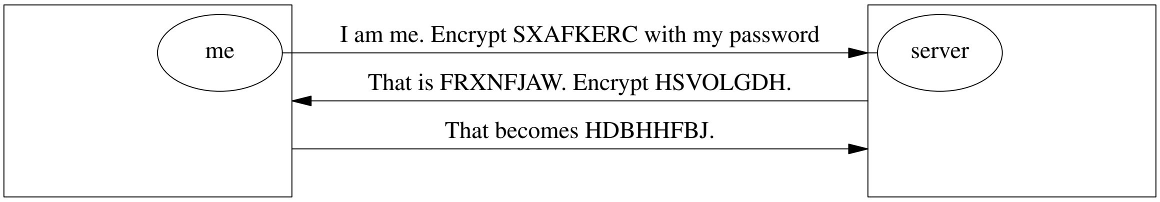
It is no better to send an encrypted version of the password; the intruder cannot find out the unencrypted password but could repeat the encrypted form to the listener without having to know the unencrypted form.

Better proposal

The previous system required that the connecting process knew the remote user's password but that it need not be held unencrypted on the machine to which he is connecting.

If both the connecter and the server (which initially has the powers of the super-user) can find out the user's unencrypted password, each can verify that the other is who he claims to be without giving any information away to an imposter.

With the first message the connecter sends a random string of characters which both processes encrypt with the user's password. The listener sends back the encrypted text and another random string which they both encrypt. The connecter then sends the encrypted version of this back to the listener.



By this means, both sides can verify that the other knows the user's password without the passage of the passwords themselves in any form. A prospective intruder can present the server with chosen plaintext and receive correctly encrypted ciphertext back but cannot correctly encrypt the server's message.

As an added measure, let the user whose files are to be accessed explicitly have permitted the caller to access his files. (Maintain a list saying which remote users from which hosts are permitted to access each person's files.) If an unsuccessful attempt is made to connect to a server, because the server encrypted some text but did not receive a correct reply (for any reason including termination of the transport service connection), that permission is revoked until the target user explicitly permits it again. This defuses a repeated chosen plaintext attack on the password.

The encryption algorithm for this must be such that it is impossible to deduce the key from one plaintext/ciphertext pair given full knowledge of the algorithm, or that an exhaustive search to find it would take several hundred years. The DES algorithm supplied with UNIX fills these requirements.

Efficiency

Extra cost is incurred by a remote execution in setting up the connection between machines, which will be greater for the Remote File System as it has to perform validation. Extra processing has to be done on each machine for each remote procedure call; commands and their arguments have to be linearised for transmission over the bytestream and decoded when received.

The need to add a few bytes of information to a (say) 1024-byte block of data to be read or written will reduce the advantage of block-buffering if the transport service used the same size of buffer. A possible remedy to this is to tune the buffer size used by the user program to a figure such that the transmission of one buffer of data results in a transport service message equal to or slightly less than its buffer size. Unfortunately, some programs rely upon their buffer size being a power of two. A safer solution would be to increase the size to (say) 4096 bytes.

The higher level of traffic on the network is not a problem at Kent as our Cambridge ring is working well below its capacity.

The need to keep more binary files, recompiled to do Remote Procedure Calls will use more filespace. Local and remote versions of the same program running on the same machine occupies more memory that two instances of the same program. However, centralising compilation, text formatting and other large tasks means that only one copy of the sharable text segment of the object code need be resident in the whole system's collective memory. What is more, the ability to even out the workload over several machines also raises the system's throughput of work.

System Dependencies

Throughout the design of this project I have tried to make it as independent as possible upon the transport mechanism and the version of UNIX used. The primitives in *trans.c* depend heavily upon using virtual circuit transport service ports, and the fake system calls correspond to those of 4.2BSD UNIX. The fake system calls for a different version of UNIX, and an implementation of *trans.c* using, say, the ‘‘sockets’’ of 4.2 BSD UNIX should be interchangeable.

A program written for 4.2 could remotely be executed on a machine running 4.1 using the 4.2 fake system calls. (A library of compatibility functions would also have to be used to map unreplaced 4.2 functions into their 4.1 equivalents.)

It is also quite feasible to implement a RFS server in an operating system other that UNIX. It would have to present a UNIX system call interface to the alien system’s filestore.

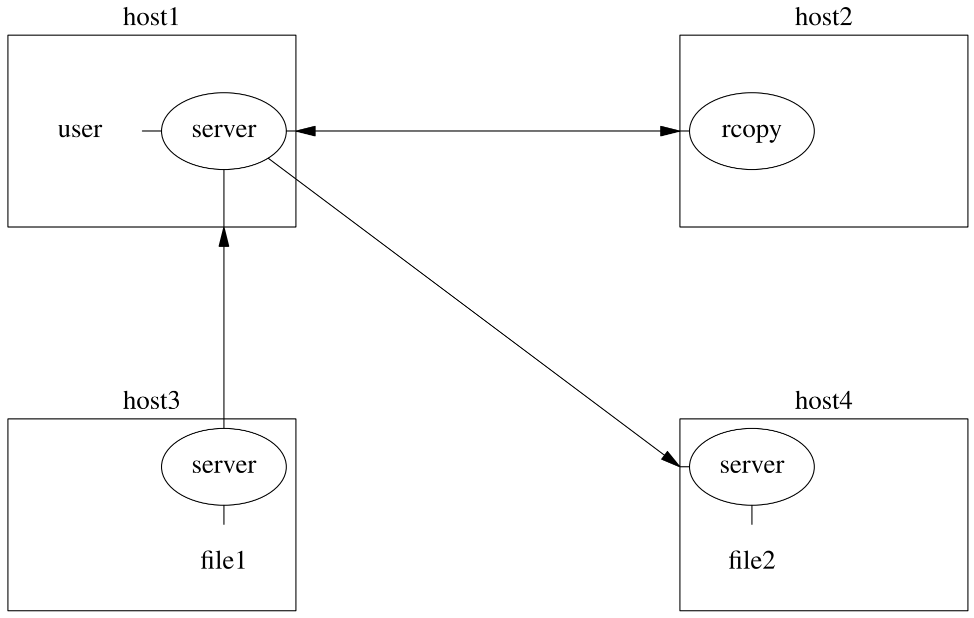
Conclusion

As a topic, Remote Procedure Calls and transparent networking are not thoroughly explored. The subject of this project is a good one. The goals are simply and clearly defined but there are many problems to be confronted in producing a satisfactory solution. The surprisingly small amount of software I produced to implement a fully working, if simple remote execution facility is the tip of a mountain of thought.

The solutions to difficult issues such as signal handling and performing forks and execs transparently have manifested themselves in the basic design of the system. For this reason it has been difficult in this report to justify some decisions as they were mentioned.

But those decisions, I think, were good. The remotely executing versions of standard, unmodified system programs such as **echo**, **printenv**, **cat** and **ls** behave identically to their local counterparts. I implemented only the system calls necessary for these test programs as I found them to be necessary; there are several dozen calls to be replaced for a fuller system for which implementation will be a simple job of imitation. Only a few such as **dup2** will require a little thought.

I am confident that I have produced a system whose generality and flexibility will make it easy to extend, and which will indeed enable large programs such as **troff** to be executed remotely. The generality is such that I see no reason why a program should not undergo loading with both RFS and REX to produce a program which both accesses files on many machines and executes remotely!



rcopy file1@host3 file2@host4

References

[1] Computer Networks, Andrew S Tanenbaum, Prentice-Hall

[2] Transport Service Byte Stream Protocol, I N Dallas, UKC Computing Laboratory Report No 1

[3] The UNIX time-sharing system, D M Ritchie and K Thompson, CACM 17 No 7 (July 1974), pp 365-375

[4] The C Programming Language, B W Kernighan and D M Ritchie, Prentice-Hall

[5] Reliable Remote Calls for Distributed UNIX: An Implementation Study, F Panzieri and S K Shivastava, Proceedings of the Second Symposium on Reliability in Distributed Software and Database Systems, July 1982