Connecting *UNIX*† Systems Using a Token Ring

Robbert van Renesse
Andrew S. Tanenbaum

*Dept. of Mathematics and Computer Science*
*Vrije Universiteit*
*Amsterdam*

Sape J. Mullender

*Centre for Mathematics & Computer Science*
*Amsterdam*

**ABSTRACT**

As part of the research on distributed operating systems being done at the Vrije Universiteit, we have implemented a set of network-oriented programs for use on several *UNIX* machines connected by a high-speed token ring. With these tools it is possible to transfer files between machines, log in to remote machines, and implement multimachine shell scripts. The transaction protocols discussed in another paper at this EUUG meeting are used to implement two basic services: a “shell server” and a data transfer service. Other services are easily implemented as shell scripts that use these services. A file transfer program, for instance, executes the command “*to < file1*” on one machine, and “*from > file2*” on the other machine. More examples of these facilities and their implementation and performance are discussed in the paper.

1. Introduction

At our university we are developing a distributed operating system called *Amoeba* [Tanenbaum81]. As a spin off from this research, we have incorporated some of the *Amoeba* interfaces into *UNIX*, and used these interfaces to build some application programs for communicating between *UNIX* systems. These tools include file transfer, remote execution and remote login. In this paper we describe the different layers into which our implementation is divided, and the interfaces that connect them, and discuss the performance of our implementation.

When we started this project we had 2 PDP11/44’s running *UNIX* V7, 2 VAX 750’s running Berkeley 4.1BSD and 8 Intel 8086’s and 8 Motorola 68000’s running *Amoeba* 1.0. As *Amoeba* was designed to be a distributed system, we needed a network.

Our network had to be fast, even under heavy load, so a ring network seemed the best choice. After some study, we chose ProNET. This is a 10 Mbit/sec star shaped

† *UNIX* is a Trademark of Bell Laboratories.
PDP, VAX and UNIBUS are registered trademarks of Digital Equipment Corporation.
ProNET is a trademark of Proteon Associates, Inc.
ring network with decentralized control and token arbitration, supporting up to 255 hosts. It can send and receive packets concurrently, do scatter/gather operations, has variable length packets up to 2044 bytes, checks parity, and has a primitive hardware acknowledgement bit. Pronet interfaces exist for UNIBUS and MULTIBUS; both are used in our machines.

Our desires, with respect to UNIX, were modest. We did not want to make a distributed system, but only some capabilities to do file transfer and remote execution. In retrospect, we feel that we have achieved these objectives.

2. Network Interface

Network application programs need a mechanism to communicate reliably. We have designed a network interface that is simple to use, which uses a efficient, simple and fast protocol. We envision communication between two processes, one is called the server and the other the client. A server handles requests from clients. When the server has handled the request it sends a reply back to the client; the sending of a request to the server and a reply back to the client is called a transaction* [Mullender84].

The transaction primitives are:

```c
typedef struct Mref {
    char *M oob;
    char *M buf;
    unsigned M len;
} Mref;
```

The client, in order to do a transaction calls

```c
trans(cap, req, rep);
    Cap *cap; Mref *req, *rep;
```

The server receives requests and sends replies with

```c
getreq(port, cap, req);
    Port *port; Cap *cap; Mref *req;
putrep(rep);
    Mref *rep;
```

3. User Programs

And now the moment of truth: can the primitives we designed be used to make useful programs? The basic things we want are file transfer and remote execution. In this section we will discuss some of the programs we have built; they fulfilled our desires and are now among the most-used programs on our UNIX systems.

MULTIBUS is a trademark of Intel, Inc.

*Not to be confused with the concept "atomic transaction."
3.1. File Transfer

The first thing expected of a fast local network is fast file transfer. We have made two simple programs to accomplish basic data transfer, requiring the user to be logged in on both the machine producing the data, and the machine consuming the data. Their syntax is:

```
from identifier
to identifier
```

To reads from standard input and from writes to standard output. If the identifiers of to and from are the same, the input data to to becomes the output data of from.

For example, when "to hamlet < /etc/passwd" is executed on machine A, and "from hamlet > /etc/passwd" on B, the password file of machine A is copied to the password file of B. The same can be done with the execution of "rcp A!/etc/passwd B!/etc/passwd," called at any machine on the network; rcp will be treated in a later section.

3.2. Remote Login

As programmers are lazy, they do not like to walk from terminal to terminal to work on different machines, especially if the terminals are in different rooms, floors or buildings. So a desire existed to be able to login onto any computer from any terminal; therefore, we made our own version of the cu command to call another UNIX system, except that our version does not lose characters. The syntax is:

```
call machine-name
```

After calling this program you get a login message from the remote machine, and you can login and work onto that machine as if the terminal is connected directly to the new machine, with one exception: lines beginning with a "~" are special. Their meaning is as follows:

```
~.: switch back to local machine;
~!: shell escape;
~: send a "~" to the remote machine;
% take from [to]: copy file "from" to local machine;
% put from [to]: copy file "from" to remote machine.
```

To execute call you will have to be logged in on some machine. If you are not, you can login as "remote." Instead of a shell you get a program that asks you for the machine you want to login on, and then executes call.

So each terminal is effectively connected to each machine. At the moment, if you inspect what each user is doing in our department, you will notice that half of them are executing call. It is useful because most of our machines are dedicated to one or two specific projects, and most of the faculty members are working on projects on different machines.

3.3. Remote execution

Many times you just want to execute a simple command at a remote machine without going to the trouble of logging in; e.g., you want to know if you are still in the top 10 of your favourite game on a certain machine, and if you are not you will have to login on this machine to fight for your place. Commands are executed on a remote machine with:

```
rsz machine command
```
The output of the command is default to the user's terminal, but can be redirected in the usual way, the input comes from "/dev/null." For example, "rsh A who" will give you a listing of the person's who are logged in on machine A.

It is now possible to run your programs on multiple machines. For example, if you want to run an neqn/nroff job, you could run it on two machines as follows:

```
(neqn file | to format)&
rsh machine "from format | nroff -ms" > out
```

The nroff output is redirected to the file "out" on the local machine. If you want to direct input to the remote machine, and split standard output and error output, you could do something like this:

```
rsh machine "from input | command | to output" >&2 &
from output&
to input
```

This means: execute command at the remote machine, with input from the process "from input" and output to "to output." Locally a "from output" is started in the background to catch the standard output of the command; the standard input is sent to the remote machine with "to input." The error output is done by the rsh process. If you put all this in a shell script, you can execute a command as if it runs locally. In the special case that this command is "sh -i," you can almost work on the remote machine as if logged in there.

### 3.4. Other Useful Programs

Out of the basic elements of file transfer and remote execution many interesting programs can be built. In this section we will discuss the programs used most on our machines; all these programs are shell scripts. Many of these scripts call to and from, which need a unique identifier as argument; for this purpose, the program uniqport outputs a random string of printable characters, to used as argument to from or to. The presented implementations of the programs are slightly simplified.

For file transfer it is a nuisance to have to login on two machines; therefore, we made a shell script called rcp which transfers files from any place in the network to any other place. Its syntax is:

```
rcp [machine1!]file1 [machine2!]file2
```

This will transfer the first file to the second. One can leave out the machine part if the file is on the local machine. An implementation, in which the machine parts are non-optional, could be:

```
IFS=! port='uniqport'
(set $1; rsh $1 "cat $2 | to $port")&
(set $2; rsh $1 "from $port | cat > $2")
```

A program related to rcp is rcat, with syntax:

```
rcat [-] [machine!]file ...
```

and obvious meaning.

An implementation of this command, with exactly one file argument, could be:
IFS=!
set $1
case $# in
  1) cat $1 ;;
  2) rsh $1 cat $2 ;;
  *) echo "Usage: $0 [machine!|file] >&2 ;;
esac

Here is another thing about programmers: they are nosy. They want to know where their fellow-programmers are logged onto, and what they are doing. For this purpose we created the programs ruho and ruw, which give information about the whereabouts and actions of all person logged in on any machine.

In our department we have several different printers attached to several different machines. Some produce ugly output fast, others produce pretty output slowly. It would be nice to print a file on an appropriate printer, independent of the machine the printer is attached to, or the system the file is on. With the program rpr you can do the same as with lpr, but with the advantages of location independence:

    rpr printer [file ...]

Its implementation, in a configuration having two printers on the machine called "tjalk" and one on the machine "klipper," is:

case $printer in
  tjalk) mac=tjalk com=lpr ;;
  pmds) mac=tjalk com=opr ;;
  klipper) mac=klipper com=lpr ;;
  *)    echo "$0: unknown printer" >&2; exit 1 ;;
esac

    port='uniport'
    rsh $mac "from $port | $com" &
    shift
    pr -t $@ | to $port ;;

Each shell script was written in 1 to 15 minutes; the basic elements of our network utilities (from, to and rsh) have proved their strength.

3.5. Implementation

Now having described the communication programs and the shell scripts we have built with them, we will discuss how from, to, rsh and call are implemented; in particular, we will take a look at the servers needed. All these programs use transactions as communication mechanism.

The implementation of from and to is simple: from acts as a server waiting for request to output data to standard output, to acts as a client doing transactions requesting the from process to output the data to has read. The port used in the transaction header is just the identifier given as argument to to and from.

To execute a command on a remote machine, a server is needed that awaits a request and executes it when one arrives. The rsh command is nothing but a client process doing a transaction with this server, requesting a command to be executed, and awaiting a reply saying the command has been executed. The servers on the different machines listen to different ports; given a machine's name, rsh knows the port to use*.

*The port is a function of the machine's name.
For remote login one also needs a server. Although the server for remote execution could be used for this purpose too, a new one is made. A simple-minded implementa-
tion of call could be the following:

```
  rsh machine "from input | sh -i" &
to input
```

The problem here is that the remote shell has pipes for input and output; for ex-
ample, you can not do ioctl's, or send signals along pipes. Therefore, we installed a de-
vice driver implementing a "pseudo terminal." The job of the remote login server is to
manage these pseudo terminals.

A pseudo terminal really consists of two devices: a master and a slave device. The
master device can be opened by a process simulating the terminal by writing to it for
terminal input, or reading from it for terminal output; the slave device just looks like
a terminal device to UNIX. The master device is called "/dev/ptyXX," and the slave
device "/dev/ttyXX." The slave device is put in "/etc/ttys" as the other terminals
are, so a getty process can manage it. The master device has two processes driving it:
the first writing to it simulating the pseudo-keyboard, and the second reading from it
simulating the pseudo-printer. These processes are just from and to, so that the
pseudo terminal can be controlled at the local machine. All the remote login server
does when it gets a request, is pick a free pseudo terminal and start the from and to
processes.

The client process call sends a message to the server requesting for a pseudo termi-
nal, sets the local terminal in RAW mode, and starts a from and a to. The from
catches the output from the pseudo terminal, and the to will send its input to the
pseudo terminal. Call just copies its input to the to process via a pipe, except for the
lines beginning with a "\", for which it must do some local processing.

As an example of how this mechanism works, we will consider what happens when
the user types a DEL character, with the intention to generate an interrupt at the
remote machine. First, the DEL is read by the local terminal driver, but because it is
working in RAW mode, it just passes the character to the reader: the call process.
Call outputs it in the pipe, giving the DEL to the to process, which sends it to the
remote from process; from writes it to the controlling site of the pseudo terminal de-
vice. Now the DEL character is treated as if the pseudo terminal was an ordinary ter-
minal where a DEL was typed in: an interrupt is sent to all the processes belonging to
the process group of this terminal.

Although the characters typed in when executing call pass through a pipe, are sent
to and echoed by the remote machine, and thus sent over the network twice, they are
sent back to the terminal fast enough to see only a delay in the exceptional case of a
lost packet, when the corresponding character has to be retransmitted. All the net-
work programs are fast enough to work with, even by impatient programmers; but
their success is mostly because of the simplicity of usage.

4. Performance

In this section we will give some performance figures for the rates we achieve using
from and to. They were measured during the middle of the day, i.e., many persons
were logged in, of whom some were working. Running the tests on a single user sys-
tem sometimes doubles the data rate, but these figures are not of any importance,
since in practice the systems are always multiuser. On the other hand, the perfor-
ance drops fast if the systems are heavily used. The rates, as shown in Fig. 1, are
not bad compared to most other systems.
<table>
<thead>
<tr>
<th></th>
<th>VAX 750</th>
<th>PDP 11/44</th>
</tr>
</thead>
<tbody>
<tr>
<td>VAX 750</td>
<td>25,000</td>
<td>15,000</td>
</tr>
<tr>
<td>PDP 11/44</td>
<td>15,000</td>
<td>10,000</td>
</tr>
</tbody>
</table>

Fig. 1. Data transfer rates in bytes per second over ProNET from user process to user process. The VAX's run 4.1BSD, and the PDP's V7. The buffer size is 512 bytes.

When we made the buffer size 2048 bytes on the VAX's, we achieved a data rate of 90,000 bytes per second (without file I/O). Unfortunately we could not use this size in general as we could not enlarge the buffer size on the PDP's.

As it does not matter where you run the network software, you may also run from and to on the same machine. The rates we achieve now are in Fig. 2. As these rates are the same as when running from and to locally, we may conclude that ProNET is not the bottleneck, but either the protocol or UNIX. Since our protocol is light weight, it must be UNIX. Indeed, when we look at where the most time is spent, it is in copying the user buffer to a kernel buffer, and in setting the timers.

<table>
<thead>
<tr>
<th></th>
<th>VAX 750</th>
<th>PDP 11/44</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>25,000</td>
<td>10,000</td>
</tr>
</tbody>
</table>

Fig. 2. Local rates. From and to both run on the same machine, and do not use ProNET.

References

[Mullender84]

[Saltzer80]

[Tanenbaum81]