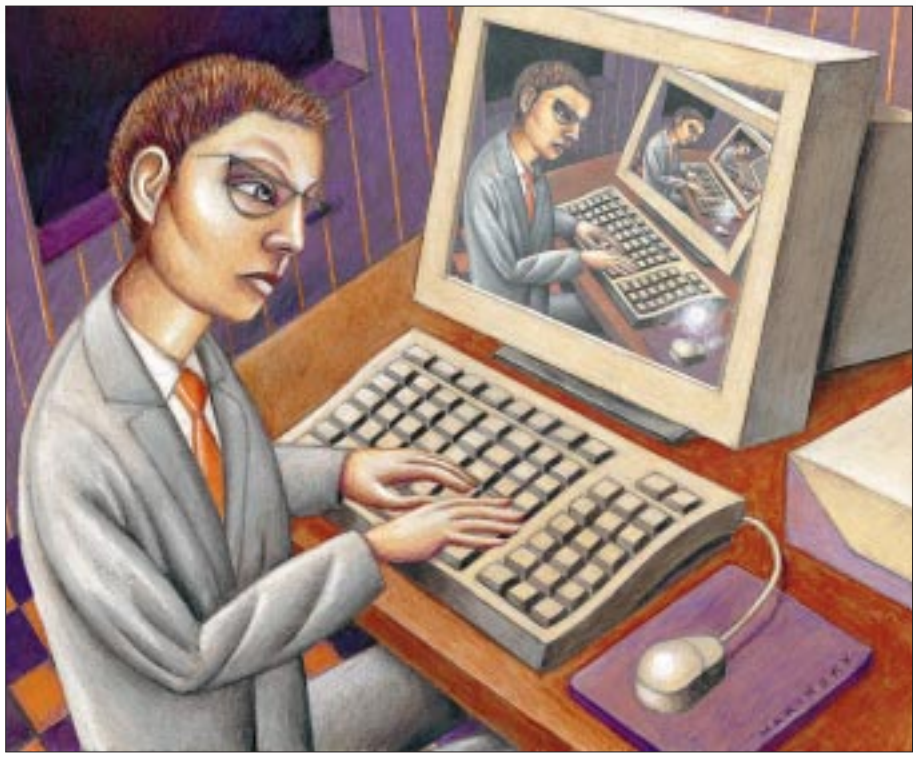
by Peter Collinson, Hillside Systems



JANE MARINSKY

The rlogin Program

I've been idly watching Robert X. Cringely's "Glory of the Geeks" television series that was recently broadcast on this side of the pond. Just in case the series passed you by, it was four programs from PBS on the history of the Internet. "Pop" histories of things with which you've been involved are always fun, providing a sort of alternate misreality. Actually there were several interesting interviews in this series, but I think that a veil needs to be quickly drawn over the things that were missed or misreported. I guess history is a winnowing process, people who really did the work get lost, while one or two make it into the history books or, in this case, onto the television screen.

Anyway, one slogan popped up in the series that's worth thinking about when defining points in the history of computing: Sun Microsystems Inc.'s slogan, "The Network is the Computer." This slogan takes us back to the beginning of Sun in the early '80s and the creation of the workstation. Of course, Sun was not actually responsible for the idea, but was the first company to make a huge success of selling a workstation that would run UNIX and opened up a market for what we now think of as the desktop computer.

Networking on UNIX started early with the development of UUCP, which began life as a quick hack connecting two machines together with serial lines and grew into a worldwide network containing several thousand machines. What's often lost in most popular versions of the Internet story is how UUCP widened expectations of how networking should be done. The UUCP network gave email and news to many people and created a demand for the fully interactive services that we use today.

The basic commands that were supported on the 4.2BSD operating system and shipped with the early Suns were also quick hacks. The `rlogin` (remote login) and `rcp` (remote copy) commands were quickly put together by Bill Joy when the networking code in the BSD kernel was newly operational. (Bill Joy, a Sun founder and software genius, was the prime mover of UNIX at the University of California at Berkeley, in case that name has passed you by.) The commands were intended as a stopgap to see if the networking code worked and were to be used only until the "official" ARPANET protocols of FTP (for file copying) and Telnet (for remote login) were developed. Of course, in the way of all quick hacks, we still have the `rlogin` and `rcp` commands on machines today and they are still useful.

rlogin

The `rlogin` command allows you to sit on one machine and log into another. You probably do this every day and take it for granted, but there's actually a considerable amount of clever stuff going on to make this work the way you want it to, which means "with as few surprises as possible." Some of the clever code resides in the program that is running locally and behaving like a terminal, which is the `rlogin` program; and some on the

remote system. Let's start with what is happening locally.

The fewest number of surprises will occur when the behavior of the remote machine when accessed over the network is identical to the normal behavior when logging in via a terminal. I suppose that "logging in via a terminal" is rare these days, but this is still the model that is used.

Terminals are bidirectional devices, that is, they permit you to type into the computer and receive text at the same time. The job of `rlogin` is to make a connection to the remote machine and wait for you to type something so it can send it down the network wire to your programs running remotely. At the same time, it needs to wait for data from the remote machine so that it can write it onto your screen. Both data flows need to be independent of one another.

The consequence of the bidirectional flow is that `rlogin` needs to look for data from two sources. Making a single program wait for data from more than one source was not easy in the early UNIX systems. The traditional approach was to split the program into two processes, each handing flow in one direction. So one process would handle data transmission from your keyboard to the remote machine, where most of its time would be spent waiting for keyboard input. The other process would handle data travelling in the reverse direction, where most of the time it would be waiting for data to come from the network.

UNIX makes all of its devices into files, so each process is forced by the kernel to wait for input from an open file. The process performs a read system call on the open file and is put to sleep. The read call returns data to the process, waking it up only when data is present in the kernel. When data appears, there's a quick flurry of activity as it is passed on and then the process goes back to sleep, waiting for more data.

Using two processes for bidirectional flow does work, but there are cases where the two processes need to share data and sharing is easier if we can use a single process instead of two. The fundamental problem is that we can only wait for data arriving from one source. The 4.2BSD solution was to implement the `select` system call. Specifically, the system call actually allows a program to wait for data to be available from one or more open files. When data arrives in the kernel from one source, the program is awakened and notified that it can now obtain data. It then reads the data, does the necessary processing and returns to wait in the `select` system call for more data to arrive. Armed with the `select` system call, we can write a program that handles two-way communication with a remote machine.

Terminal Characteristics

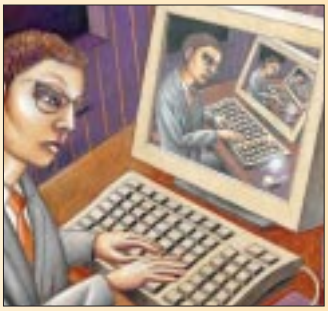
We must now think about the characteristics of terminals. When using a regular terminal, a user types a character that's sent into the host machine and is echoed back to appear on their screen. There should be minimal delay when echoing the character (the delay is the apparent response time of the machine). When we are connecting to the machine via a regular terminal and using command-line programs, the kernel will do the work of echoing each character back to the screen. The kernel hangs onto the data and handles line editing, such as the deletion of the last character or the last word. It will wait for the Return key to be typed before sending the completed line of text to the user process that is waiting for the data.

However, there are several "visual applications" where the process needs the input immediately without the Return key being hit–the process wants the data sent one character at a time. Also, it may turn echoing off because a single keystroke may translate into a complex command for the editor. UNIX has developed ways of permitting a program to take such control of terminal input and output. The UNIX terminal interface is now a complicated piece of code, controlled by the processes on the machine. Settings in the interface depend on the application the user is using.

If we are writing a program like `rlogin` (or `telnet`), we have a choice about where character echoing can be done. Option one is to echo the characters locally, using local kernel processing to handle things such as character deletion. When the user types Return, a line of clean data can be sent to the remote machine as a single complete message. Option two is to echo the characters from the remote machine. So when a user types a character, it's not echoed locally but rather sent to the remote machine and echoed back from there. Input is usually done one character at a time and if the network is slow, user response time can be poor. Also, a packet is sent over the network every time a user types a character, so the amount of network traffic is much higher and the network is being used more frequently.

Option one works fine when the user is typing lines of information and was the method used by default in early `telnet` programs that ran over very slow lines. On slow networks it meant that the user didn't have a perceptible delay after each character was typed. The response was fast because the local machine was doing all the work. However, the method begins to break down when the user is logged into a remote system and wants to run a program such as a visual editor that normally takes control of the keyboard and screen. These programs will execute some special system calls to configure the terminal interface and, in general, there is no provision made to communicate that change of state in the terminal interface on the remote machine to the user's `rlogin` or `telnet` program running locally.

Because rlogin is designed to work over a LAN, there are no problems with unsatisfactory response times owing to the passing of single characters. In fact, the problems of using single-character I/O over the Internet have diminished with time.



There's no reason why one cannot construct a protocol that will allow terminal interface state to be passed over the network so that all terminal processing can take place locally. In fact, the X.29 protocol, which supported terminal access over the X.25, did permit the passing of terminal control state. However, what happened at the terminal interface was very operating system-specific and because X.29 was designed to work in heterogeneous environments, it inevitably didn't support all the features needed to provide good transparent access to a UNIX system.

It's actually much simpler to handle all the "funny" terminal processing on the remote machine and make the local emulator work in one-character-at-a-time mode. This means the local `rlogin` program simply has to pass each character through in either direction as fast as it can, while all the clever stuff is done remotely.

Because `rlogin` is designed to work over a local-area network (LAN), which should be fast, there are no problems with unsatisfactory response times owing to the passing of single characters. In fact, the problems of using single-character I/O over the (much wider area and slower) Internet have diminished with time. Nowadays, I regularly make a 6,000-mile electronic commute from Canterbury to Berkeley and it works fine, supplying more than adequate response time. I suppose I should qualify that a little: It's fine until someone on the East Coast of the United States decides in the middle of the night that it's safe to reconfigure that router, at which point my line drops out until the network reroutes itself.

If you place no intelligence about character handling in the local program, there remains one final set of problems that are worth mentioning. On a UNIX system, we expect to be able to type a character and have a program die on the machine. On most systems, typing Control-C will stop a running process dead, so that if a program is writing reams of stuff to your screen, it will stop and the system will become usable again. With the `rlogin` character-passing approach, the local program passes your Control-C character back to the remote host, which interprets the character as an interrupt signal and stops your process. This approach works, but there may be data in transit and you may get more data on the screen than you bargained for.

OK. So we understand what's needed on the local machine, and I'll bet that it's a more complicated story than you thought. What about the remote machine? What do we need to run to make things work?

The Remote Machine

Well, our first problem on the remote machine is that all the processes that we need to run for the user are designed to talk to terminals supported by the complex plethora of system calls used to establish terminal state. However, we are planning to communicate with the remote machine with network packets. We need to find some method of changing the data from the incoming network packets into data that appears on a system device that behaves like a terminal. Similarly, when the user process writes data, it will do so thinking that it's writing to a terminal. We want to capture that data, translate it into network packets and transmit it to the locally running `rlogin` process that started everything off.

I was faced with this problem in the early '80s. Our campus plan was to connect terminals to concentrator boxes (running on z80s for the historically minded). The concentrators would reach out over the local area Cambridge Ring-based network and communicate with the UNIX machine. I implemented a terminal device driver that fed data from the network into the regular UNIX terminal-handling code (and vice versa). User processes would talk to some code that behaved like a terminal, except that the data was being transmitted to the network and onto a remote terminal rather than being passed along a serial line to some directly connected VDU.

The folks at Berkeley took a more general approach and wrote a special-purpose driver called a *pseudo-terminal* or *pseudo-tty*. Each invocation of the device driver has two ends in the file system address space: the terminal end, which behaves like a regular terminal, and the control end, which provides traditional UNIX data streams in both directions.

When the local `rlogin` connects to the remote machine, a daemon (`rlogind`) is started whose first task is to find a free pseudo-tty and initiate an appropriate login process on the terminal end of the device. The job of `rlogind` is to sit in a loop waiting for data from the network and stuffing it down the control end of the pseudo-tty. It also waits for messages from the control end (actually output from the user process to its terminal) and sends them over the network to the

local machine. By the way, I am trying to retain consistencyhere. For clarity: The local machine is the machine on which the user is typing and the remote machine is running `rlogind`.

Actually, the mechanism is fairly inefficient. Data from the user's keyboard goes into the local kernel and passes across the system call interface into the local `rlogin` program. The data is immediately parcelled up and sent back across the system call interface and out onto the network. Each data packet arrives in the kernel of the remote machine, where it passes across the system call interface into `rlogind`, which in turn immediately passes it back into the control end of the pseudo-tty. It then travels across a system call interface once more and finds itself in the user's process. Of course, the user's process will immediately reply with some information that is sent back along the tortuous route to the user. We've all got enough CPU power now and don't really notice all this data yo-yoing around between user processes and the kernel. Back when I was designing my system, I didn't want to take this approach because I was trying to support in excess of 50 simultaneous users on a 1-MIP VAX11/780 and the possible load was a big concern.

Nevertheless, the general-purpose nature of the pseudo-tty interface was and is a win. It is useful for other programs that wish to start user processes. For example, my preferred editor, JOVE, uses a pseudo-tty to implement a command that runs a user shell in an editor window.

Telnet

As I said at the top of the article, `rlogin` was not supposed to have endured as long as it has. It should have been replaced by `telnet`. The replacement hasn't happened. People still use `rlogin`. The issue is probably one of convenience. The `telnet` command is designed to support heterogeneous machines and, therefore, is not as well tailored to the UNIX-to-UNIX login application. UNIX has always wanted to know the type of its terminals so that it can adapt visual programs to send the correct control sequences to effect cursor addressing on the target screen. When `rlogin` connects to a remote machine it sends the terminal type in a secret prologue so the remote machine can set things up for you. Of course, `telnet` doesn't have this mechanism and so you need to establish the terminal type to enable visual editors to work properly.

The `rlogin` program is also supported by an ad hoc authentication system that permits automatic login to the remote machine without needing to supply a user name and password. When you login, your identity depends on the name of the local machine, determined by reverse lookup from your IP address to a machine name (it also depends on the user name on the local machine that is passed in the secret prologue).

Originally, the integrity of the authentication system depended ultimately on the fact that `rlogin` connected to the remote machine from a "privileged port number," a port number less than 1,024. Recall that a TCP/IP stream connects to an interface that has an IP address and a port on that interface. In a UNIX-only world, it was possible to stop users from creating communication paths using the set of privileged ports, however, once the PC gained networking code all this false security went away.

Current security wisdom states that you should not run `rlogin` over the Internet because it's too easy to break its weak security. In fact, I have a set of filters on my router that prevents anyone outside my network from using `rlogin` to access an internal machine.

The convenience and speed of `rlogin` can be regained by augmenting the somewhat weak, old system with Kerberos. I tend to use `ssh` these days when connecting over the Internet so that the whole communication is encrypted, protecting my passwords against packet sniffers and other systems used by the bad guys.

PS. I've since discovered from reading *Wired* magazine that Cringley's series is called "Nerds 2.0.1: A Brief History of the Internet" on your side of the Atlantic. Why there should be a name change as the tapes travel over water is beyond any reasoning. I am sure that we would understand "Nerds," and I cannot believe that "Geeks" would not have worked for an American audience. Perhaps TV programs need longer names in North America to stand out in the TV listings books, which are generally physically smaller than ours. Who knows.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* `pc@cpg.com`.