Interprocess Communication in the Eighth Edition Unix System

D. L. Presotto
D. M. Ritchie

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

When processes wish to communicate, they must first establish communication, and then decide what to say. Previously described stream mechanisms of the Eighth Edition Unix system[1] provide a flexible way for processes to speak with devices and with each other. An existing stream connection is named by a file descriptor, and the usual read, write, and I/O control requests apply. Processing modules may be inserted dynamically into a stream connection, so network protocols, terminal processing, and device drivers separate cleanly.

This paper describes ways of commencing communication. The traditional, and simplest, is the pipe. In our system, pipes are just cross-connected streams.

A new request associates a stream with any named file. When the file is subsequently opened, operations on the file are operations on the stream. Also, open files may be passed from one process to another over a stream, and opening a stream-associated file may create a new and distinct channel to the stream's server.

These low-level mechanisms allow considerable flexibility in constructing network dialout routines and connection servers of various kinds.

Introduction

The Eighth Edition version of Unix is the system that runs on machines in the Information Sciences Research Division of AT&T Bell Laboratories, and at a few sites elsewhere. It is named, by our custom, after its manual.

The work reported here provides convenient ways for programs to establish communication with unrelated processes, on the same or different machines. The kind of communication we are interested in is conducted by means of ordinary read and write calls, occasionally supplemented by I/O control requests. Moreover, we wish to commence communication in ways that are as close as possible to ordinary file opening. These considerations spring from the desire to find ways of fitting even complicated things into a simple pattern, and from the observation that whenever objects behave like files, practically any program is able to use them.

In particular, we study how to

1) provide objects nameable as files that invoke useful services, such as connecting to other machines over various media,
2) make it easy to write the programs that provide the services.

Unix is a trademark of AT&T Bell Laboratories

Recapitulation

The Eighth Edition system introduced a new way of communicating with terminal and network devices,[1] and a generalization of the internal interface to the file system.[2,3] We begin by reviewing already-published nomenclature and mechanisms of our I/O and file systems.

Streams

A *stream* is a full-duplex connection between a process and a device or another process. It consists of several linearly connected processing modules, and is analogous to a Shell pipeline, except that data flows in both directions. The modules in a stream communicate by passing messages to their neighbors. A module provides only one entry point to each neighbor, namely a routine that accepts messages.

At the end of the stream closest to the process is a set of routines that provide the interface to the rest of the system. A user's *write* and I/O control requests are turned into messages sent along the stream, and *read* requests take data from the stream and pass it to the user. At the other end of the stream is either a device driver module, or another process. Data arriving on the stream is sent to the device or read by the process; data and state transitions detected by the device, or generated by the process, are composed into messages and sent into the stream towards the user program. Intermediate modules process the messages in various ways. They are symmetrical; their read and write interfaces are identical.

The two end modules in a stream to a device become connected automatically when the process opens the device; streams between processes are created by a *pipe* call. Intermediate modules are attached dynamically by request of the user's program. They are addressed like a stack with its top close to the process, so installing one is called 'pushing' a new module.

Queues

Each stream processing module consists of a pair of *queues,* one for each direction. A queue comprises not only a data queue proper, but also two routines and some status information. One routine is the *put procedure,* which is called by its neighbor to place messages on its data queue. The other, the *service procedure,* is scheduled to execute whenever there is work for it to do. The status information includes a pointer to the next queue downstream, various flags, and a pointer to additional state information required by the instantiation of the queue. Queues are allocated in such a way that the routines associated with one half of a stream module may find the queue associated with the other half. (This is used, for example, in generating echoes for terminal input.)

Message blocks

The objects passed between queues are blocks obtained from an allocator. Each contains a *read pointer,* a *write pointer,* and a *limit pointer,* which specify respectively the beginning of information being passed, its end, and a bound on the extent to which the write pointer may be increased.

The header of a block specifies its type; the most common blocks contain data. Control blocks of several kinds have the same form as data blocks and are obtained from the same allocator. For example, there are control blocks to introduce delimiters into the data stream, to pass user I/O control requests, and to announce special conditions such as line break and carrier loss on terminal devices.

Examples

Figure 1 shows a stream device that has just been opened. The top-level routines, drawn as a pair of half-open rectangles on the left, are invoked by users' *read* and *write* calls. The writer routine sends messages to the device driver shown on the right. Data arriving from the device is composed into messages sent to the top-level reader routine, which returns the data to the user process when it executes *read*.

Figure 2 shows an ordinary terminal connected by an RS-232 line. Here a processing module

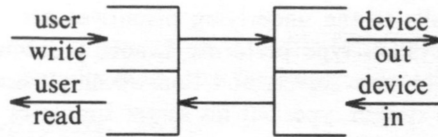


Figure 1. Configuration after device open.

(the pair of rectangles in the middle) is interposed; it performs the services necessary to make terminals usable, for example echoing, character-erase and line-kill, tab expansion as required, and translation between carriage-return and new-line.

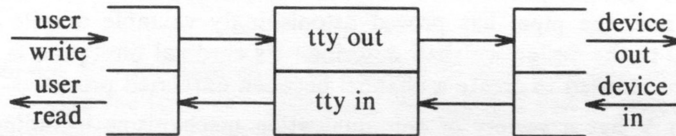


Figure 2. Configuration for normal terminal attachment.

Many network connections require flow- and error-control protocols to be carried out by the host computer. Therefore, when terminals are connected to a host through such a network, a setup like that shown in Fig. 3 is used; the terminal processing module is stacked on the network protocol module.

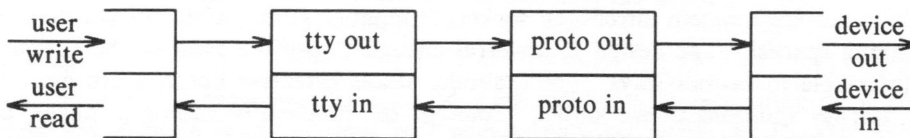


Figure 3. Configuration for network terminals.

A common fourth configuration (not illustrated) is used when the network is used for file transfers or other purposes when terminal processing is not needed. It simply omits the "tty" module and uses only the protocol module. Sometimes, on the other hand, a front-end processor conducts the required network protocol. Here a connection for remote file transfer will resemble that of Fig. 1, because the protocol is handled outside the operating system; likewise network terminal connections via the front end may be handled as shown in Fig. 2.

Finally, Figure 4 shows the connections for a pipe. In our system, pipes are full-duplex.

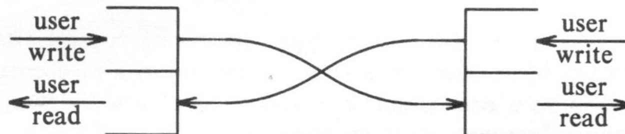


Figure 4. A pipe.

File Systems

Weinberger[2] generalized the file system. He identified a small set of primitive operations on inodes (read, write, get, put, truncate, get status, etc.: a total of 11) and created a form of the *mount* request that specifies a file system type and, where appropriate, a stream. When file

operations are requested, the calls to the underlying primitives are routed through a switch table indexed by the type. His file system type performs remote procedure calls across the associated stream, at the other end of which is a server, and thus accomplishes a remote file system. Pike[4] takes advantage of the same file system type, but his server simulates a disk containing images classified by machine, person's name, and resolution.

Killian[3] added a file system type that appears to be a directory containing the names (process ID numbers) of currently running processes. Once a process file is opened, its memory may be read or written, and control operations can start it or stop it.

Establishing communication

Traditional Unix systems provide few ways for a process to establish communication with another. The oldest one, the pipe, has proved astonishingly valuable despite its limitations, and indeed remains central in the design we shall describe. Its cardinal limitation is, of course, that it is anonymous, and cannot be used to create a channel between unrelated processes.

AT&T's System V has a variety of communication mechanisms including semaphores, messages, and shared memory. They are all useful in certain circumstances, but programs that use them are all special-purpose; they know that they are communicating over a certain kind of channel, and must use special calls and techniques. System V also provides named pipes (FIFOs). They reside in the file system, and ordinary I/O operations apply to them. They can provide a convenient place for processes to meet. However, because the messages of all writers are intermingled, writers must observe a carefully designed, application-specific protocol when using them. Moreover, FIFOs supply only one-way communication; to receive a reply from a process reached through a FIFO, it is necessary to construct the return channel explicitly.

Berkeley's 4.2 BSD system introduced sockets (communication connection points) that exist in domains (naming spaces). The design is powerful enough to provide most of the needed facilities, but is uncomfortable in various ways. For example, unless extensive libraries are used, creating a new domain implies additions to the kernel. Consider the problem of adding a 'phone' domain, in which the addresses are telephone numbers. Should complicated negotiations with various kinds of ACUs be added to the kernel? If not, how can the required code be invoked in user mode when a program calls 4.2's *connect* primitive?

The Eighth Edition's variant file systems, mentioned above, provide a general way of establishing communications between processes. Indeed, they are a bit too general for many problems; it requires a sophisticated server to simulate a file system. Therefore, we tried to find simpler, but still general, ways of connecting programs.

Additions

We made three additions to the system.

Mounted streams

First is a new, but very simple, file system type. Its *mount* request attaches a stream named by a file descriptor to a file. Most often the stream is one end of a pipe created by the server process, but it can equally well be a connection to a device, or a network connection to a process on another machine. Subsequently, when other processes open and do I/O on that file, their requests refer to the stream attached to the file. The effect is similar to a System V FIFO that has already been opened by a server, but more general: communication is full-duplex, the server can be on another machine, and (because the connection is a stream), intermediate processing modules may be installed.

By itself, a mounted stream shares the most important difficulty of the FIFO; several processes attempting to use it simultaneously must somehow cooperate.

Passing files

The second addition is a way of passing an open file from one process to another across a pipe connection. Although they are actually done with *ioctl* operations, the primitives may be written

```
sendfile(wpipefd, fd);
```

in the sender process, and

```
(fd1, info) = recvfile(rpipefd);
```

in the receiver. The sender transmits a copy of its file descriptor *fd* over the pipe to the receiver; when the receiver accepts it, it gains a new open file denoted by *fd1*. (Other information, such as the user- and group-id of the sender, is also passed.)

Unique connections

Finally, we found a way for each client of a server to gain a unique, non-multiplexed connection to that server. It takes the form of a processing module that can be pushed on a stream, which will usually be mounted in the file system as described above. When the file is opened by another program, this module creates a new pipe, and sends one end to the server process at the other end of the mounted stream, using the same mechanism as the *sendfile* primitive described above. After the server has called *recvfile* to pick up its end of the pipe, it may accept or reject the new connection; if it accepts, the other program's *open* call succeeds, and its open file refers to the local end of the new pipe to the server. If the server rejects the request, the *open* fails.

Examples

A graded set of examples will illustrate how to use these facilities.

Network calling

Originating network connections is a complicated activity. There is often name translation of various kinds, and sometimes negotiations with various entities. With our Datakit VCS network,[5] for example, a call is placed by negotiating with a node controller. When dialing over the switched telephone system, one must talk to any of several kinds of ACU devices. Setting up a connection on an Ethernet under any of the extant protocols requires translation of a symbolic name to a net address. These protocols should certainly not be in kernel code. It is usual to put setup negotiations in user-callable libraries, but it is better to have all the code for each network in a single executable file. In this way, if something in the network interface changes, only one program needs to be fixed and reinstalled.

A program desiring to make a connection calls a simple routine that creates a pipe, forks, and in the child process executes the network dialer process. The dialer either returns an error code, or passes back a file descriptor referring to an open connection to the other machine. The pseudo-code for the library routine, neglecting error-checking and closing down the pipe, is:

```
netcall(address)
{       int p[2];
        pipe(p);
        if (fork()!=0)
                execute("/etc/netcaller", address, ascii(p[0]));
        status = wait();
        if (bad(status))
                return(errcode);
        passedfd = recvfile(p[1]);
        return(passedfd.fd);
}
```

The `/etc/netcaller` program can be arbitrarily complicated. Its job is to create the connection and either fail, returning an appropriate error code, or succeed, and pass its descriptor for the open

connection. Along the way, it may negotiate permissions and provide the caller's identity reliably, because it can be a privileged (set-uid) program.

Process connections

Suppose you are writing a multi-player game, in which several people interact with each other through a controller process. It might be a banker (Monopoly) or a mazekeeper (Mazewar) or a tournament director (Bridge). The problem is to set up a single process prepared to receive asynchronous connection requests from new players. In our solution, there are two programs: the controller, set up initially, and the player program, executed by users as they enter the game. When the controller starts, it creates a pipe, pushes the unique-connection processor on one end of the pipe, mounts it on a conventionally-named file (say `/tmp/gamester`), and waits for connection messages to arrive. When the player program is run, it opens `/tmp/gamester`, thereby doing an implicit *sendfile*. The controller notices that there is input on its connection pipe (perhaps making use of *select*) and accepts the connection with *recvfile*. Thereafter, the player program transmits moves and receives replies over the file descriptor obtained from opening the gamester file, and the controller reads the player's moves and transmits replies over the file descriptor it received.

Network calling (advanced course)

The network calling routines described above can be generalized by writing a connection server. The mechanism is illustrated in Figure 5. In structure, the connection server is exactly like the game master; it maintains a conventional file (say `/server`) with a stream mounted on it (5a).

Users of the service (programs like *cu* and *rlogin*, or our local addition to the family, *dcon*), open `/server` and write on it the name of the entity they wish to contact (5b). The server program reads the name, and translates it to a true address and the name of one of the network caller programs described above. It then executes the caller program (5c) and passes the resulting connection file back to *dcon* (5d). Finally, *dcon* accepts the new connection and closes the connection stream (5e).

A fine point in the design is the decision to make the connection server a continuously-existing process, and to communicate the desired address by writing on its mounted file. Another possibility is to invoke it like the caller programs in the first example: by executing it, with the user's address as a parameter. From the user's viewpoint, these two interfaces is equally effective. We are trying the current approach because it seems more efficient, in that the mapping tables can be cached in the server process, and it requires fewer process creations. It is also a more interesting experiment in program design, because it is so highly multiplexed.

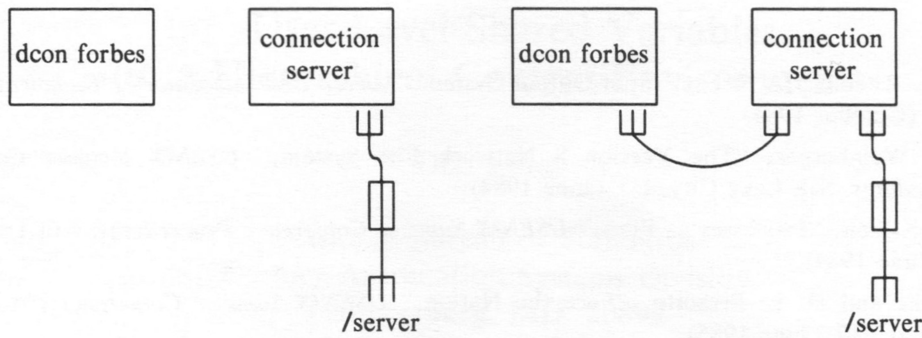
The structure encourages experimentation with naming plans. A domain setup seems natural to try. Suppose we have domains *dk*, *inet*, and *att* that refer respectively to Datakit, Internet, and the switched telephone system. Then the following address translations, which yield a complete Datakit address, Internet host address, and telephone number, are appropriate for our gateway machine:

```
dk/research      →     mh/astro/research
inet/research    →     192.11.4.55
att/research     →     2015825940
```

There are, of course, complications: the machine is on two ethernets that have to be distinguished; various line speeds have to be specified when dialing with an ACU. Finally, it is necessary to coalesce the *dcon*, *rlogin*, and *cu* programs.

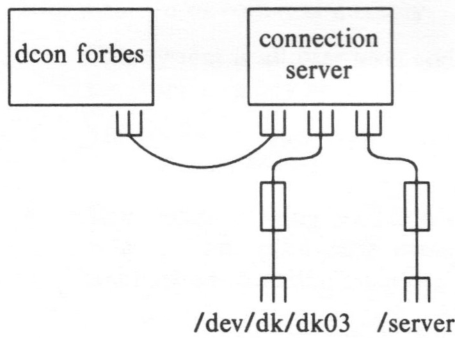
Conclusion

Unix has always had a rich file system structure, both in its naming scheme (hierarchical directories) and in the properties of open files (disk files, devices, pipes). The Eighth Edition exploits the file system even more insistently than its predecessors or contemporaries of the same genus. Remote file systems, process files, and the face server all create objects, the name of which can be handed as usefully to an existing tool as to a new one designed to take advantage of the object's special properties. Similarly, the stream I/O system provides a framework for making

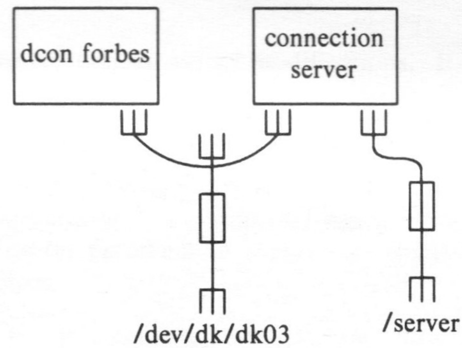


(a) Server mounts a stream onto /server.

(b) *dcon* command opens /server.



(c) Server sets up network connection.



(d) Server passes connection to *dcon* command.

(e) Process accepts network stream and closes stream to server.

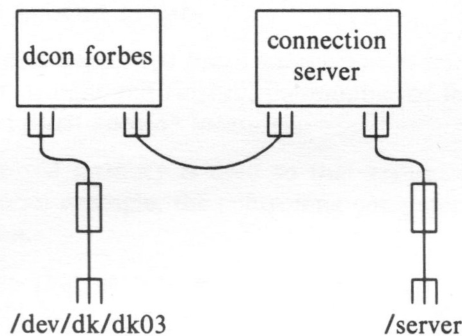


Figure 5. Establishing a network connection.

already opened files behave in the standard way most programs already expect. For example, the real purpose of the terminal-processing module is to mediate between programs expecting a simple byte stream, and imperfect typists using terminals with peculiar control requirements. A network protocol module's purpose is to make an error-prone network, again with idiosyncratic properties, conform to a simpler model.

The developments described here follow the same path; they encourage use of the file name space to establish communication between processes. In the best of cases, merely opening a named file is enough. More complicated situations require more involved negotiations, but the file system still supplies the point of contact. Moreover, the necessary negotiations may be encapsulated in a common form that hides the differences between local and any of a variety of remote connections.

References

1. D. M. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal* **63**(8) (October 1984).
2. P. J. Weinberger, "The Version 8 Network File System," *USENIX Summer Conference Proceedings*, Salt Lake City, UT (June 1984).
3. T. J. Killian, "Processes as Files," *USENIX Summer Conference Proceedings*, Salt Lake City, UT (June 1984).
4. R. Pike and D. L. Presotto, "Face the Nation," *USENIX Summer Conference Proceedings*, Portland, OR (June 1985).
5. A. G. Fraser, "Datakit—A Modular Network for Synchronous and Asynchronous Traffic," *Proc. Int. Conf. on Commun.*, Boston, MA (June 1980).