Communication Files:
Interprocess IO before Pipes

M. Douglas McIlroy

Dartmouth College
doug@cs.dartmouth.edu
February, 2017

Introduction

Some time after the introduction of pipes to Unix, we in the Bell Labs Unix lab learned that the Dartmouth Time-Sharing System (DTSS) had a mechanism for process-to-process IO called *communication files*. Unfortunately we didn't know exactly how they worked. When I retired from Bell Labs to Dartmouth in 1997, I asked around fruitlessly for further information. At last, at a DTSS reunion organized by Tom Kurtz, who had fostered the project, I met Sidney Marshall, who had been involved in the implementation. He explained the concept. The picture was rounded out in discussion with another participant, Stephen Garland, who had edited the *DTSS Programming Manual*.

Communication files were much more complicated than Unix pipes. They were also more powerful. Pipes could be simulated by communication files, but not vice versa. A pipe can handle neither the two-way communication nor the out-of-band signaling that communication files support.

The programming manual's description of communication files ran to many pages. [available at http://www.cs.dartmouth.edu/~doug/DTSS/DTSSchapter5.pdf] As a result, communication files remained beyond the working toolkit even of many DTSS insiders. Nevertheless communication files played an indispensable role: one or more communication files mediated every user's interaction with the system.

Much of the detail below comes from collections of DTSS documents that Garland and Marshall have deposited with the Dartmouth library. Further information was gleaned from an email conversation among DTSS alumni, to which Peter Doyle kindly introduced me.

Dates

Communication files significantly antedated Unix pipes. Evidence from design documents puts the origin of the concept sometime in 1967, between the writing of an outline of features for the Phase II DTSS system dated March, which doesn't mention communication files, and a summary of executive services dated 29 August, which does. As communication files were used for terminal sessions, they were operational when the system went live on January 6, 1969 [John Kemeny, January 20, 1969]. (More than three years before pipes debuted in UNIX.) A DTSS glossary from the time contains the description,

‘‘Communications [sic] file − A type of file organization which allows direct communication with a job in the system rather than with an input/output device.’’ The facility was described briefly in a 1969 conference session about DTSS:

> A communications file allows two jobs to interact directly without the use of secondary storage. A communications file has one end in each of two jobs. It is the software analog of a channel-to-channel adaptor. This structure allows job-to-job interactions using the same procedures as for more conventional files. The two ends are labeled master end and slave end. A job at the slave end of a communications file cannot easily distinguish this file from a conventional file. **Since a job at the master end of a communications file can control and monitor all data transmitted on that file, a master end job can simulate a data file, thereby providing a useful debugging aid and also providing a convenient mechanism for interfacing running jobs to unexpected data structures.** [my emphasis; Robert F. Hargraves, Jr. and Andrew G. Stephenson, ‘‘Design considerations for an educational time-sharing system’’, AFIPS Spring Joint Computer Conference 1969, pages 657-664]

Dim reflections of the insight in the highlighted sentence began to appear in Unix-family systems in the 1980s. The full concept finally took hold as a guiding principle in Plan 9, still without awareness of this early formulation.

Functionality

Communication files gave complete control of the open-file API to a user process. The concept, which Sidney Marshall and other DTSS alumni attribute to Ken Lochner, [http://www.cs.rit.edu/swm/history/DTSS.doc] was motivated by the intent to handle the details of terminal sessions outside the kernel, in accord with the policy that an ‘‘absolute minimum of the executive system was written to run in master-mode, so as to make debugging and modification of software as easy as possible.’’ [Kemeny, ibid] The Hargraves/Stephenson paper describes the terminal-handling mechanism in considerable detail.

In full generality, communication files supported synchronous and asynchronous data transfer, random access, status inquiries, out-of-band signaling, error reporting and access control in addition to the primary read, write and close operations. Within this broad outline, the semantics of the API was determined by each individual master process.

A process could set up a communication file, hold on to one end—the ‘‘master’’—and pass the other ‘‘slave’’ end to a descendent process. Data transfers were always initiated at a slave end. The master end, alerted by interrupt, would match a slave write with one or more master reads.

A process could acquire a slave end in two ways. The slave end could be inherited by a newly created child process, in which case the child could be oblivious to the fact that it was dealing with a communication file. Alternatively, the slave end could be transmitted by a PASS operation executed by a process that had access to the communication file. The target process had to overtly prepare to receive a PASS, which came via interrupt.

Usage

A notable application of communication files was in support of *conferences*, which behaved somewhat like conference phone calls. Conferences were a service of SIMON (SImple MONitor), the primary user interface to the system, like a shell in Multics or Unix. A conference was created by a LINK operation, which started an arbitrary program to manage the conference. Other user sessions could subsequently JOIN the named conference link. The program managing a conference did not hold the master ends for the conference. That function was performed by a program called MOTIF (Multiple On-line Terminal InterFace), which handled arrivals and departures and gathered communication into a single multiplexed data stream to the conference manager. [John McGeachie, ‘‘Multiple terminals under user program control in a time-sharing environment’’, *CACM* **16** (1973) 587-590]

Among the uses of conferences were multiperson games, and online course registration. One conference, which we would now call a chat room, ran essentially continuously for some 15 years.

Comparative success of communication files and pipes

Why did communication files attract little notice in the computing community, while Unix pipes had lasting influence?

A major reason is simply the huge spread of Unix, which caught attention for its simplicity, utility and low cost. Yet even at home in Dartmouth, communication files were used for only a few specific applications, while pipes became part of every Unix programmer’s toolkit. Unix’s command-line combinator ‘‘|’’ fostered the habit; nothing in DTSS did. The question then becomes why not?

A facile answer is that Unix pipes and the pipe combinator were created as (almost) inseparable twins. By contrast, Lochner’s grand concept stood nearly alone, abetted only by MOTIF for conferences. Communication files became familiar to very few people. There was a forbidding amount of detail to learn. The potential barrier between the mechanism and a simple use like pipes was quite high.

A similar phenomenon can be seen in the rarity in Unix of pipe topologies other than simple chains. Nothing like MOTIF has gained purchase to suppport games and analogous applications. Although many shells, including the very popular *bash*, have offered some facilities for conncting processes in tree- and even dag-shaped topologies, the capability has barely been exploited. One deterrent is the lack of a standard convention for passing open IO connections beyond stdin and stdout, which might enable fancier plumbing. Perhaps Diomidis Spinellis’s recent and quite comprehensive *dgsh* has a chance of injecting dag connections into the vernacular. [http://www.spinelllis.gr/cs/dgsh.html]

Using communication files, it would have been easy to implement a pipe combinator in SIMON. I have not heard that the possibility was ever discussed. Even if it had been proposed, a pipe would effectively have been like a two-party conference coordinated by a pass-through process in the middle. The prospect of increasing the population of active processes might have been worrisome. It is also likely that some motivating factors were uncommon in the largely student environment: (1) multistep computations made of cascading programs and (2) a library of discrete utility programs beyond compilers, editors and word processors, available for combining.

Ahead of its time?

Pipes simply enabled interprocess IO—a small variation on the preexisting Unix model. Communication files were a concept of a different order—a lifting of the file API to user-level implementation. In this, it was more akin to Plan 9's 9P protocol than to familiar IO. Shoehorned into the IO model, instead of being engineered from the ground up, commuunication files became complicated beyond necessity for the purpose of piping, but not comprehensive enough to enable a completely fresh approach to distributed computing as Plan 9 would eventually do.

Had it been widely known, the underlying idea of separating the file interface from its implementation in order to enable alternate implementations might have inspired Plan 9-like efforts earlier. As events actually transpired, Plan 9 attracted considerable interest, and some of its surface features were promptly incorporated into other systems, while its central principle had little influence. Twenty years after Plan 9 and nearly fifty after DTSS, the incumbent mechanisms of distributed computing remain largely unaffected by either. By the time Plan 9 offered the distilled essence of communication files, the momentum of the old model seems to have become too great to deflect.