Portals in 4.4BSD

W. Richard Stevens
Consultant

Jan-Simon Pendry
Sequent UK

Abstract

Portals were added to 4.4BSD as an experimental feature and are in the publicly available 4.4BSD-Lite distribution. Portals provide access to alternate file types or devices using names in the normal filesystem that a process just opens. For example, an open of `/p/tcp/foo.com/smtp` returns a TCP socket descriptor to the calling process that is connected to the SMTP server on the specified host. By providing access through the normal filesystem, the calling process need not be aware of the special functions necessary to create a TCP socket and establish a TCP connection. This makes TCP connections, for example, available to programs such as Awk, Tcl, and shell scripts.

This paper describes the implementation of portals in 4.4BSD as another type of filesystem and provides some examples.

1. Introduction

The Unix paradigm of `open`, `read`, `write`, `lseek`, and `close` has worked well for many years across a wide range of devices. Having devices share the same namespace as regular files is one of the many things Unix did right. Terminals and networks, however, have always been unique and present an interesting set of problems.

One problem with terminal devices is the wide variety of modems that connect to them, and each modem's unique command string. This has led to the duplication of dialing code among various programs, such as `tip`, `cu`, and UUCP.

The problem with networks is that they have their own unique set of functions to create and manipulate network end points. Whether using the sockets programming interface or TLI, the code to establish a network end point differs greatly from the code for normal file I/O [Stevens 1990]. While the C programmer can easily access network connections using either sockets or TLI, it has been a problem to access network connections using scripting languages such as a shell or Awk. One solution, chosen by Perl [Wall and Schwartz 1991] is to place all the socket calls into the language. While possible, this has the unfortunate side effect of increasing the size and complexity of the language.

The implementation of portals in 4.4BSD is an attempt to hide the complexity of things such as network connections within a system-wide portal daemon that is accessible to *any* process through the `open` function.

2. Descriptor Passing

Modern versions of Unix have supported *descriptor passing* between unrelated processes for many years. 4.2BSD was the first system to support this, circa 1983, although there were bugs in the initial implementation, which were fixed with 4.3BSD (1986). System V Release 3.2 (1988) also provided a similar capability, and a cleaner interface was provided with System V Release 4 (SVR4). Most commercial versions of Unix today, whether derived from BSD or System V, provide this capability. [Stevens 1990; 1992] provides examples of descriptor passing in both the BSD and System V worlds.

3. Connection Server

With the ability to pass an arbitrary descriptor between unrelated processes the idea of a *connection server* came about [Presotto and Ritchie 1985; 1990]. This server operates as follows:

- Any process creates a connection to the server, called a *stream pipe*. 4.3BSD implements stream pipes using Unix domain stream sockets, while the `pipe` system call is used under SVR4.
- The server receives notification that a new client has connected to it and a unique connection is created for each client. The BSD `accept` function

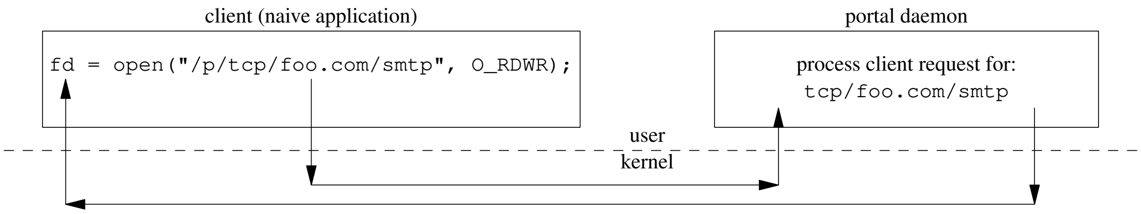

Figure 1: Overall design of portals.

creates the unique connections, as does the SVR4 `connld` streams module.

- The client sends a request to the server, often by writing some data to the stream pipe that the server reads.
- The server validates the request and does whatever is required. This might involve establishing a network connection to a remote system, or opening a modem device and dialing a remote system, for example. The purpose of the connection server is to place all these communications-dependent operations into a central user-mode server instead of scattering the same operations through numerous applications, or instead of trying to place them into the kernel.
- The server uses descriptor passing to return a descriptor to the client. The client can `read` and `write` this descriptor, without involving itself in all the steps required to open the descriptor.

[Stevens 1992] provides a complete example of a connection server that dials a modem, along with some general purpose client–server connection functions that hide the differences between the BSD and SVR4 implementations. The `ipc`(3X) man page [AT&T 1990] documents another set of client–server connection functions, the ones described in [Presotto and Ritchie 1990]. As powerful as the concept of a connection server is, it has been used sparingly in the widely used releases of Unix.

One problem with the connection server model is that it cannot be used by *naive* applications. That is, applications that only call `open`, `read`, and `write`. To take advantage of a connection server the application must be coded to connect to the server, send a request, and receive a descriptor in reply. Connection servers cannot be used by scripting languages such as Awk, Tcl, and shell scripts.

4. Portals

Portals provide a generalized "open" capability, similar to a connection server, but they solve the problem of naive applications. Portals are accessed by the normal `open` function. Since most applications (be it a scripting language or a binary application that we want to run but do not have the source code for) allow us to enter a pathname of our choice, and since portals are accessed by pathnames, this makes portals available to virtually *any* application.

For example a naive application calls `open` with `/p/tcp/foo.com/smtp` as the pathname argument and the kernel returns either a nonnegative descriptor or −1 if an error occurs. The application then `reads` and `writes` the descriptor. Figure 1 shows the basic design.

The pathname argument to `open` begins with the *mount point* of a portal filesystem (`/p` in all the examples in this paper). The kernel passes this to the portal daemon, as we describe in detail in the next section. The daemon does whatever it needs to for this client, which in this case involves calling `gethostbyname` to convert the name `foo.com` into one or more IP addresses, calling `getservbyname` to obtain the port number for the `smtp` service, calling `socket`, and then calling `connect`. The daemon either succeeds and passes a descriptor back to the client, or the client's call to `open` returns −1 with `errno` set to the appropriate error code.

There are a few key points in the implementation of portals in 4.4BSD.

1. The only function of the portal code within the kernel is to pass the pathname argument from `open` to the portal daemon, along with the client's credentials, and to take the daemon's return value, either a descriptor or an error code, and pass it back as the return value from `open`. The kernel does not interpret the pathname argument at all, other than detecting the portal filesystem mount point. The interpretation of the string `tcp`, followed by a host name, followed by a service name, is handled by the daemon.
2. The portal daemon is a user process that can easily be enhanced or replaced. As experience is obtained with portals, additional objects besides TCP connections can be added without changing the kernel.

3. The use of portals and the portal daemon is completely transparent to the application. As a trivial example,

   ```
   $ cat < /p/tcp/noao.edu/daytime
   Wed Jul  6 11:26:07 1994
   ```

   opens a TCP connection to the standard daytime server on the host `noao.edu`, which writes back the current time and date and closes the connection.

4. Since the client's `open` goes through the kernel before the request is passed to the portal daemon, the kernel provides the client's credentials to the server: the client's effective user ID, effective group ID, and supplementary group IDs. This information is passed to the daemon, in addition to the pathname from `open`. This lets the server implement any type of access restrictions that it desires, based on the client's identity.

5. Related Work

Ideas similar to portals have appeared in numerous operating systems over the past decade. We mention these briefly here since space prevents a detailed comparison of the various implementations.

The 4.2BSD manual [Joy et al. 1983] defined the `portal` system call, with seven arguments, and a footnote that it was not implemented in 4.2BSD.

[Bershad and Pinkerton 1988] describe *watchdogs*: user-level processes that can be called whenever a specified file or directory is opened.

The Sprite operating system provides pseudo devices [Welch and Ousterhout 1988]. It allows a user-level process to emulate a file or device.

Plan 9 [Presotto and Winterbottom 1993] represents network connections using *protocol devices*. All protocol devices look identical so user programs contain no network-specific code.

6. Implementation of Portals

Portals are implemented as a new filesystem type within the 4.4BSD kernel. When the system is initialized a portal daemon is normally started. The daemon performs the following steps:

1. Creates a Unix domain stream socket, which we call `listenfd`.
2. Calls `listen` to allow incoming connection requests to be accepted on the socket.
3. Calls the `mount` function to mount the portal filesystem at a specified location in the filesystem (the mount point), typically `/p`. Additional portal-specific arguments to the `mount` function are the socket descriptor of the listening socket created by the server, and the server's process ID.
4. The portal daemon then goes to sleep in an `accept` function, waiting for some process to open a pathname that begins with the portal mount point.

These four steps are summarized in Figure 2.

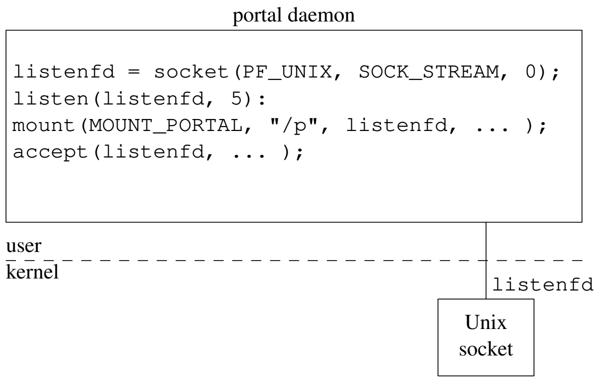

Figure 2: Initialization of portal daemon.

Within the vnode architecture of 4.4BSD (similar to the vnode architecture described in [Welch 1994]), the portal filesystem provides 11 filesystem (or *vfs*) operations: mount, unmount, statfs, and so on. About 40 file (or *vnode*) operations are also defined for each filesystem type: open, close, lookup, and so on.

Whenever a process (which we call the client) opens a pathname that begins with the portal mount point, the kernel passes control to the vnode lookup function defined for the portal filesystem: `portal_lookup`. The following steps take place.

1. `portal_lookup` allocates a new `vnode` structure and the remainder of the pathname (whatever follows the mount point) is saved in the structure. For example, if a process opens the pathname `/p/tcp/foo.com/smtp`, the string `tcp/foo.com/smtp` is saved in the `vnode` structure.
2. This new vnode is returned by `portal_lookup` and it becomes an argument to `portal_open`, which is the next kernel function called to process the `open` function.
3. `portal_open` creates two new Unix domain stream sockets: the first we call `clifd`, and the second we call `connfd`. The second of these sockets is created and returned as the new

descriptor by the `accept` pending on the portal daemon's listening Unix socket. The first socket (`clifd`) is connected to `connfd`, as though both had been created by the `socketpair` function. This creates a full-duplex pipe between `clifd` and `connfd`. Additionally, since this "pipe" is a Unix domain socket, descriptors can be passed across it.

Figure 3 shows the current state of the server. The client is blocked in its call to `open`.

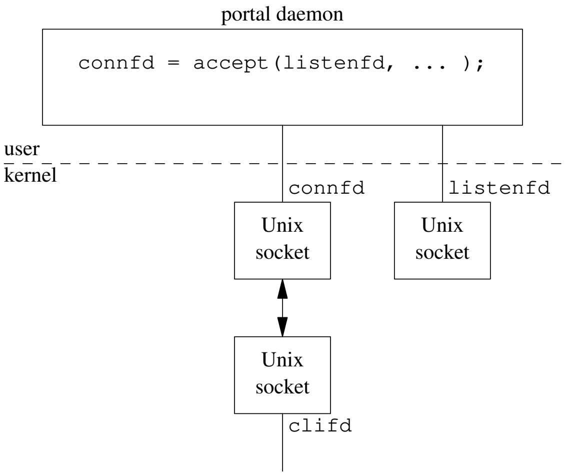

Figure 3: Client calls `open`, kernel creates two sockets and connects them together.

4. The portal daemon calls `fork`, like a typical concurrent server, and the child handles the new connection request on the descriptor returned by `accept`. The child closes `listenfd`, the parent closes `connfd`, and the parent calls `accept` again, waiting for the next client to open a file whose pathname begins with the portal mount point.

   From this point on we show only the child, since it is handling this client's request.

5. `portal_open` builds a message that is written to `clifd`, which the child reads on `connfd`. The message contains the client's credentials (a `portal_cred` structure, shown below) followed by the remainder of the pathname that was saved in the vnode by `portal_lookup` (the string `tcp/foo.com/smtp` in our example). The child can validate the client using this information.

   The structure written by the kernel is:

```
struct portal_cred {
  int   pcr_flag;     /* file open mode */
  uid_t pcr_uid;      /* effective uid  */
  short pcr_ngroups; /* # of group IDs */
  gid_t pcr_groups[NGROUPS]; /* gids   */
      /* effective gid = pcr_groups[0] */
};
```

   Figure 4 shows the current state of the server child.

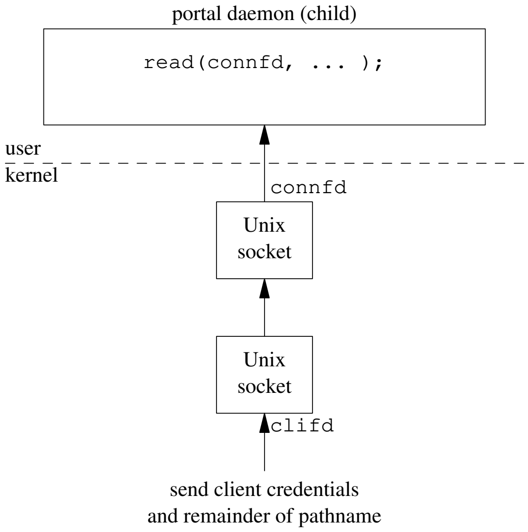

Figure 4: Kernel sends client credentials and remainder of pathname to child.

6. The child reads the `portal_cred` structure and the remainder of the pathname, and does whatever it needs to do with the client request. Notice that the client's request is implicitly specified by the pathname that the client opens. All the client has done is call `open`. The client is not aware of what's going on in the kernel, or that the portal daemon and its child are involved.

   Continuing our example, the child creates a TCP socket and connects it to the specified server on the specified host. Assuming this succeeds, we have the sockets shown in Figure 5.

7. The child calls `sendmsg` on `connfd` to pass two items back to the kernel: a 4-byte error code and optionally the descriptor `tcpfd` that becomes the descriptor returned to the client by its `open`. The descriptor is returned only if the error code is 0.

   The child passes the file descriptor to the kernel using the descriptor-passing capabilities of `sendmsg`. The descriptor is received by the

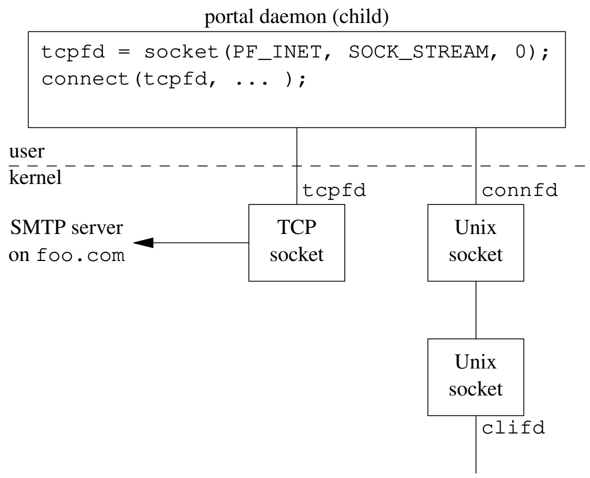

Figure 5: Child creates TCP socket and connects to specified server.

`portal_open` code in the kernel using the kernel equivalent of `recvmsg` on `clifd`. The child closes `tcpfd`, closes `connfd`, and terminates.

If a nonzero error code is returned by the child, that value is returned in the client's `errno` and the client's call to `open` returns −1. In our example the error code could be `ECONNREFUSED` (connection refused), `EHOSTUNREACH` (host unreachable), or `ETIMEDOUT` (connection timed out). These error codes extend the ones listed on the `open`(2) manual page.

Figure 6 shows the status of the various sockets when the child passes the TCP socket to the kernel.

8. If a descriptor is passed by the child (the error code is 0), it is converted into the lowest unused descriptor in the client process and becomes the return value from `open`. We call this `tcpfd1`. Its integer value will probably differ from the value of `tcpfd` in the child, but both of these descriptors refer to the same file table entry within the kernel. This is a basic property of passing descriptors: the descriptor value probably differs between the sender and the receiver, but both descriptors refer to the same file table entry in the kernel. Figure 7 shows the status of the TCP socket in the client, after the client's call to `open` returns.

The 4.4BSD implementation of portals requires about 1,000 lines of C code within the kernel. The portal daemon is another 1,200 lines of C code. Additional objects added to the portal namespace increase the size of only the portal daemon. The size of the kernel routines does not change.

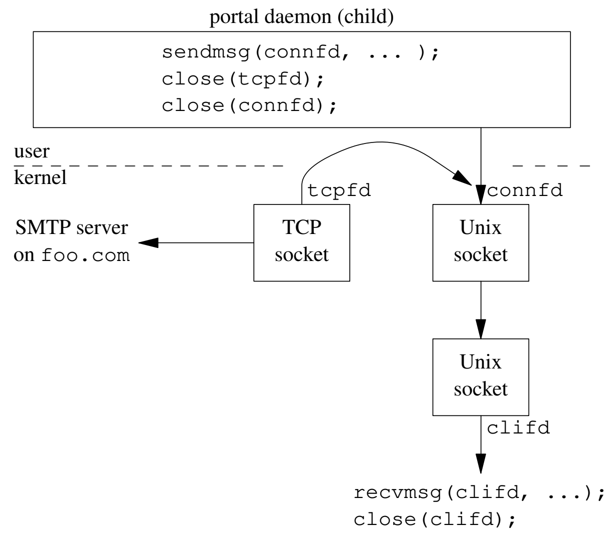

Figure 6: Status of descriptors while `tcpfd` is passed from child to kernel to client.

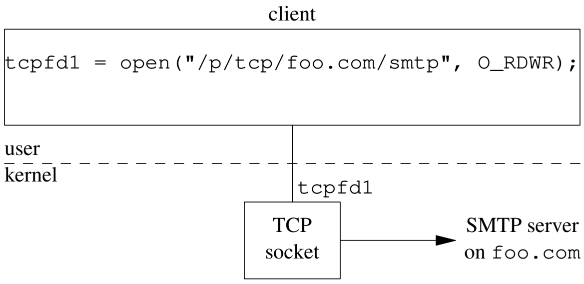

Figure 7: Final status of TCP descriptor.

7. Examples

7.1. TCP Namespace: Clients

The 4.4BSD implementation of the portal daemon creates TCP client connections when the pathname consists of

`tcp`/*host*/*service*

where *host* is either a domain name (searched for by `gethostbyname`) or an IP address, and *service* is either a service name (found in `/etc/services`) or a decimal port number. For example

```
tcp/news.host.com/nntp
```

Additionally the string `/priv` can be appended to the pathname to specify that a privileged port (less than 1024) should be bound to the socket.

7.2. TCP Namespace: Servers

A TCP server is created when the pathname consists of

`tcplisten/`*IPaddr*`/`*service*

IPaddr is the IP address of a local interface bound to the server's listening socket. The kernel will only accept incoming connection requests that arrive on this interface. The special value of 0.0.0.0 (called the wildcard) allows the server to accept incoming connection requests on any local interface. *service* is either a service name or port number that specifies the server's well-known port.

There are two problems with this type of portal service. First, the portal daemon child calls `listen` and `accept`, waiting for the client connection request to arrive. If all is OK, the connected socket descriptor returned by `accept` is returned by the child and becomes the return value of `open`. The child closes the listening socket before it terminates. This means the server that calls `open` is an iterative server, and not a concurrent server. There is no way to leave the daemon child in existence, with the listening descriptor still open, waiting for additional client connections.

The TCP server could `fork` its own child to handle the new client, after `open` returns, and then call `open` again, to wait for the next client connection. But this leaves a window of time when a listening server does not exist on the server's well-known port. Any client connections arriving in this window will be actively rejected by TCP.

Second, servers normally need to know the identity of the client. A server written using sockets would call `getpeername` to obtain the client's IP address and port number once the connection had been accepted. The server can use this information to validate the client. With portals, the daemon child could call `getpeername` after `accept` returns, but there is no way to return this to the server as another return value from `open`. Unless the server is written using a language that provides access to the `getpeername` function, the client's identity is unavailable to the server.

7.3. An Echo Server and Client

[Stevens 1990] shows the C code for a TCP echo server and client. The client reads a line from standard input, sends the line to the server, reads back the echoed line from the server, and writes the echoed line to standard output. Each requires about 60 lines of C code. Here is an equivalent server in Tcl [Ousterhout 1994] using portals.

```
#!/usr/contrib/bin/tclsh

set f [open "/p/tcplisten/0.0.0.0/7654" r+]

while {[gets $f line] >= 0} {
    puts $f $line
    flush $f
}
```

The port number of 7654 is the value chosen for this client and server.

Here is the corresponding client in Tcl.

```
#!/usr/contrib/bin/tclsh

set f [open "/p/tcp/bsdi.tuc.noao.edu/7654" r+]

while {[gets "stdin" line] >= 0} {
    puts $f $line
    flush $f

    gets $f reply
    puts $reply
}
```

Here is the same client, written using the KornShell.

```
#!/bin/ksh

exec 3<>/p/tcp/bsdi.tuc.noao.edu/7654
                           # open on fd3

while read -r sendline     # from stdin
do
    print -r -u3 $sendline # to server
    read -r -u3 recvline   # from server
    print -r $recvline     # to stdout
done

exec 3<&-                  # close fd3
```

Using command-line arguments, the name of the server host and the port number could be specified at run-time, instead of being hard-coded within the various scripts.

7.4. Filesystem Namespace

When the pathname begins with `fs/` the portal daemon opens the named file, starting at the daemon's root. For example, an `open` of

```
/p/fs/tmp/temp.1
```

opens the file /tmp/temp.1, relative to the daemon's root. Since the file open mode (the second argument to open) is passed in the portal_cred structure, the daemon can open the file using the options specified by the caller.

There are two potential uses for the filesystem namespace. First, it can provide a controlled escape from a chrooted environment. Second, the portal daemon can provide access control lists (ACLs, [Curry 1992]) to allow specific users access to files to which they normally do not have access.

8. Problem Areas

8.1. Standard I/O Problems With Network Connections

In the Tcl client and server just presented, an explicit flush is performed after each write to the TCP socket. This is required since languages such as Tcl and Awk normally use the standard I/O library. On the first I/O operation after a standard I/O stream is opened, a call to the isatty function is issued by standard I/O. If the standard I/O stream (not to be confused with the streams I/O system within the kernel) is a terminal device, the stream is line buffered, otherwise the stream is fully buffered. Since a socket is not a terminal device, the stream is fully buffered. The standard I/O library will not call write until either the buffer is full (normally the buffer size is 8192 bytes) or the application calls fflush.

This is a problem because when the line of output going to the server is buffered by the client, the server will be blocked reading from its socket and the client will be blocked reading from its socket. This is called *deadlock*. This is the same problem described in Section 14.4 of [Stevens 1992] with regard to coprocesses and the standard I/O library.

8.2. Awk Problems With Network Connections

Two problems arise when using portals with the Awk language. First is the standard I/O flush problem mentioned above, because unlike Tcl, there is no way to flush an Awk I/O stream explicitly.

Second is the inability to specify the open mode of a file with Awk. (The r+ in the Tcl scripts says to open the file for read and write.) Awk opens a file read-only or write-only: there is no way to open a file for reading and writing, which is required for a full-duplex socket.

8.3. Interruptibility of Network Connections

When a process creates a TCP connection, the process is often put to sleep by the connect function, which can take seconds to complete. If the connection cannot be established, and is not actively rejected by the other end point, the call to connect normally blocks for 75 seconds. When we run a program such as a Telnet client or an FTP client, we just type our interrupt key to abort the connection attempt.

Since an open of a portal object is a single system call, and involves another process (the portal daemon), the call to open is not interruptible in the 4.4BSD implementation of portals. We can type our interrupt key as often as we like, but until the daemon returns an error after 75 seconds, the call to open cannot be interrupted. This can be frustrating to users.

9. Performance

9.1. Portals Versus Other Forms of Descriptor Passing

We ran a set of timing tests to compare the performance of portals versus other forms of client–server descriptor passing. The goal is to verify that portals do not perform worse than other forms of descriptor passing. Three tests were run.

1. Portals.
   The client program opens the pathname /p/tcp/127.0.0.1/13, the daytime server on the local host. It then reads from the descriptor that is returned, writing the result to standard output (which is redirected to /dev/null). The descriptor is closed. This loop is executed 20 times.

   The portal daemon was modified for this test, removing the fork of a child process. This is because the next two tests do not involve a fork, and it also removes the time required for the fork from the overall timing. Also, a dotted-decimal IP address and decimal port number are specified, to remove two more variables from the timings: the calls to gethostbyname (which involves a name server) and getservbyname. Finally, the daytime service is provided internally by the inetd server and does not involve a fork or an exec.

	descriptor passing (Sec. 9.1)			portal/direct open (Sec. 9.2)	
	BSD/386		SVR4	BSD/386	
	portals	Unix domain sockets	`connld`	portals	direct client
client real time	0.53	2.21	2.76	1.90	1.21
client user CPU time	0.00	0.00	0.05	0.01	0.05
client system CPU time	0.15	0.36	0.41	0.19	0.23
client total CPU time	0.15	0.36	0.46	0.20	0.28
server total CPU time	0.11	0.20	0.01	1.04	
client + server total CPU time	0.26	0.56	0.47	1.24	0.28

Figure 8: Timing comparisons: descriptor passing and portal versus direct open.

2. Berkeley Unix domain sockets.
   This test is a slight modification of the client–server functions in Section 15.5.2 of [Stevens 1992]. A server is started that creates a Unix domain socket and binds its well-known pathname to the socket. The client creates a Unix domain socket and connects to the server. The client writes a one-line request to the server, specifying the name of a file to open and waits for the server to return either a descriptor or an error code. The server was modified to always create a TCP connection to 127.0.0.1, port 13. The client reads from the returned descriptor, writing the result to standard output (redirected again to `/dev/null`). The client closes the connection to the server. This loop is executed 20 times.

   The client and server do not maintain the connection between them each time around the loop, to allow comparison with the portal version. Naturally, if the client and server maintained this connection, the continual passing of a client request followed by the return of a descriptor by the server would be faster than creating and then tearing down the connection for each request.
3. SVR4 `connld` streams module.
   This test is similar to the previous one, but it uses the client–server connection functions based on the SVR4 `connld` streams module from Section 15.5.1 of [Stevens 1992].

Tests 1 and 2 were run on the same hardware (a 25 Mhz 80386) with the same operating system (BSD/386 Version 1.1, with the portal code from 4.4BSD-Lite added by the authors). Test 3 was run on the same hardware, but a different operating system (SVR4 Version 2.0). This adds some variability to the timings because the creation of a TCP connection to the local daytime server uses different TCP/IP implementations on the two different operating systems. Also, on this hardware BSD/386 provides higher resolution timing by reading the high resolution hardware clock, while SVR4 provides timings only in multiples of the system software clock (10 ms per clock tick).

The first three columns of Figure 8 shows the timing results. All times are in seconds.

The first three rows give the real time (i.e., the wall clock time on an otherwise idle system), user CPU time, and system CPU time for the three clients. The server total CPU time was obtained by running the `ps` command before and after the client was run.

These numbers show that a Unix domain socket is about the same as the SVR4 `connld` module (which isn't surprising, since they do similar things) but the portal implementation is faster still.

9.2. Portals Versus Direct Client Open

Another set of timing tests were run to compare a client using portals versus a client calling `socket` and `connect` directly. We expect the latter to be faster, since the overhead of invoking the portal daemon and the passing and receiving of a descriptor is avoided.

1. Portal client.
   The client program opens the pathname `/p/tcp/sun.tuc.noao.edu/daytime`, the daytime server on a different host on the local subnet. A hostname and service name are used this time, instead of the numeric values from the previous set of timing tests, to include the typical overhead involved in contacting a name server and reading the `/etc/services` file. The host running the name server is another host on the local subnet. Additionally, the `fork` of a child was put back into the portal daemon, since this is its normal mode of operation. The client reads from the descriptor that is returned, writing the result to standard output (which is redirected to `/dev/null`). The descriptor is closed and the loop is executed 20 times.

2. Direct client open.
   The client calls `gethostbyname`, `getservbyname`, `socket`, and `connect`, the same steps performed by the portal daemon. The client reads from the socket, writing the result to standard output (redirected to `/dev/null` again). The socket descriptor is closed and the loop is executed 20 times.

The final two columns of Figure 8 show the timing results. All times are in seconds.

The real time for this portal version (1.90) is higher than the real time for the previous portal version (0.53). We expect this since a `fork` is occurring, and the child is calling `gethostbyname` and `getservbyname` before it can return the connected descriptor.

The server CPU time in these examples was obtained by running the `ps -S` command before and after the client was run. The `-S` flag causes the CPU time output to include the CPU time for all terminated children. We need to account for the CPU time of the 20 children, since that is where the calls to `gethostbyname`, `getservbyname`, `socket`, and `connect` take place.

The end result of these comparisons is that the portal client requires about 50% more clock time and about 4.5 times the CPU time as a direct client. Remember, however, that this test only measures the times to establish a connection with a server on the local subnet, receive about 30 bytes of data from the server, and terminate the connection. Most TCP connections are dominated by data transfer, not connection setup and teardown. Once the connection is established, the data transfer time is unaffected by whether the socket descriptor was obtained directly by the client, or from a portal daemon.

10. Summary

Portals can provide access to a variety of objects, such as devices, special file types, and network connections, to naive applications. This is done by placing a minimal amount of code within the kernel to detect special pathnames that are being opened, and pass these to a user-mode daemon. This daemon can do whatever it wants to provide access to the specified object. A descriptor is passed from the daemon to the kernel and back to the process as the return value from `open`.

Although descriptor passing and connection servers have been possible for years within major Unix systems, the use of these facilities has never been transparent. Portals are an attempt to make these facilities transparent, although portals are not without their own unique set of problems.

Portals have been implemented under 4.4BSD but could easily be implemented in any version of Unix that supports different types of filesystems.

Acknowledgments

One of the authors (jsp) is responsible for the design and implementation of portals in 4.4BSD and should be credited with the ideas herein. The other author (wrs) discovered them in the 4.4BSD release and felt they were an important concept to document for others to learn about. Our thanks to Gary Wright and the anonymous referees for their comments on this paper.

References

Aho, A. V., Kernighan, B. W., and Weinberger, P. J., *The AWK Programming Language,* Addison-Wesley, Reading, Mass. (1988).

AT&T, *UNIX Research System Programmer's Manual, Tenth Edition, Volume I,* Saunders College Publishing, Fort Worth, Tex. (1990).

Bershad, B. N., and Pinkerton, C. B., "Watchdogs—Extending the UNIX File System," *Computing Systems,* 1, 2, pp. 169-188 (1988).

Curry, D. A., *UNIX System Security: A Guide for Users and System Administrators,* Addison-Wesley, Reading, Mass. (1992).

Joy, W. N., Cooper, E., Fabry, R. S., Leffler, S. J., McKusick, M. K., and Mosher, D., "4.2BSD System Manual," UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Volume 2C, Computer Systems Research Group, Univ. of California, Berkeley, Calif. (1983).

Ousterhout, J. K., *Tcl and the Tk Toolkit,* Addison-Wesley, Reading, Mass. (1994).

Presotto, D. L., and Ritchie, D. M., "Interprocess Communication in the Eighth Edition UNIX System," *Proceedings of the 1985 Summer USENIX Conference,* Portland, Oreg. (1985).

Presotto, D. L., and Ritchie, D. M., "Interprocess Communication in the Ninth Edition UNIX System," *Software Practice & Experience,* 20, S1, pp. S1/3-S1/17 (1990).

Presotto, D., and Winterbottom, P., "The Organization of Networks in Plan 9," *Proceedings of the 1993 Winter USENIX Conference,* pp. 271-280, San Diego, Calif. (1993).

Stevens, W. R., *UNIX Network Programming,* Prentice-Hall, Englewood Cliffs, N.J. (1990).

Stevens, W. R., *Advanced Programming in the UNIX Environment,* Addison-Wesley, Reading, Mass. (1992).

Wall, L., and Schwartz, R. L., *Programming perl,* O'Reilly & Associates, Sebastopol, Calif. (1991).

Welch, B., "A Comparison of Three Distributed File System Architectures: Vnode, Sprite, and Plan 9," *Computing Systems,* 7, 2, pp. 175-199 (1994).

Welch, B. B., and Ousterhout, J. K., "Pseudo Devices: User-Level Extensions to the Sprite File System," *Proceedings of the 1988 Summer USENIX Conference,* pp. 37-49, San Francisco, Calif. (1988).

W. Richard Stevens is the author of *UNIX Network Programming* (Prentice-Hall, 1990), *Advanced Programming in the UNIX Environment* (Addison-Wesley, 1992), *TCP/IP Illustrated, Volume 1: The Protocols* (Addison-Wesley, 1994), and coauthor with Gary R. Wright of *TCP/IP Illustrated, Volume 2: The Implementation* (Addison-Wesley, 1995). He prides himself on having no degrees in computer science, instead obtaining a B.S.E. in Aerospace Engineering from the Univ. of Michigan (1973) and a Ph.D. in Systems Engineering from the Univ. of Arizona (1982). He can be reached via email as `<rstevens@noao.edu>`.

Jan-Simon Pendry is a graduate of Imperial College, London where he did postgraduate research on environments to support logic engineering. He is probably best known as the author of *Amd*, the freely available and widely ported automounter, and also as the author of six of the pseudo-filesystems in 4.4BSD. Jan-Simon has worked for IBM and CNS and is now a consultant with Sequent Corporation in Europe. You can contact him via email at `<jsp@sequent.com>`.