Extending UNIX File Abstraction for General-Purpose Networking

Padmanabhan Pillai, Kang G. Shin

IRP-TR-04-27
January 2004

Research at Intel

Extending UNIX File Abstraction for General-Purpose Networking

Padmanabhan Pillai
Intel Research Pittsburgh

Kang G. Shin
University of Michigan

Abstract

A very powerful and basic feature of Unix-like operating systems is the file abstraction they provide for most devices and system resources. This enables a wide range of applications to work with a wide range of target devices with a simple consistent API. However, in the context of networking, the file abstraction is only partially implemented, requiring a very different sockets interface for connection/channel establishment.

This paper extends and implements a complete file abstraction for general-purpose networking. We show that the most common usage models can be directly supported, and that very simple file-based utilities can be made to work as Internet connectivity tools. These powerful abstractions are available without significant cost, as the additional overheads are limited to less than 400 μs increase in connection-establishment latencies, and no more than 7% reduction in total throughput for our prototype implementation.

1 File Abstractions in UNIX

A fundamental and very powerful, consistent abstraction provided in UNIX and compatible operating systems is the file abstraction. Many OS services and device interfaces are implemented to provide a file or file system metaphor to applications. This enables new uses for, and greatly increases the power of, existing applications — simple tools designed with specific uses in mind can, with UNIX file abstractions, be used in novel ways. A simple tool, such as `cat`, designed to read one or more files and output the contents to standard output, can be used to read from I/O devices through special device files, typically found under the `/dev` directory. On many systems, audio recording and playback can be done simply with the commands, "`cat /dev/audio > myfile`" and "`cat myfile > /dev/audio`," respectively.

This very powerful file abstraction is also used for communication. Interprocess communication (IPC) in UNIX is available through a myriad of different interfaces. One basic, common mechanism is through named pipes [3], or FIFOs, which implement unidirectional communication streams. The interface for using these pipes provides a file abstraction. Pipes, like device files, are created as special files, using the `mknod` call. Once a pipe has been created, all ordinary access functions to find, open, close, read, and write to the pipe work as they do with ordinary files. Of course, since pipes are character streams and not real files, certain functions (such as seeking within a file) are not meaningful and hence unavailable. Nonetheless, once a pipe has been created, any program that uses files in a sequential manner or has its input/output redirected to the pipe can use the IPC channel.

The file abstraction is also used in networking, but this abstraction is only partially implemented. Once a network connection is established (or a port is bound for connectionless protocols), a program can use file access functions to communicate with the remote system. In particular, the `read`, `write`, and `close` system calls all can be used on established network channels. Unfortunately, the establishment of a communication channel is performed through an entirely different interface, the Berkeley Sockets API [2]. Although it is very general and can handle a wide range of protocols, the sockets interface is somewhat clumsy to use and requires that programs be written specifically to use it. This breaks away from the nice, consistent file abstractions used through the rest of UNIX.

In this paper, we extend the file abstraction to include network channel establishment as well as communication operations. We describe how file operations may be used in place of network-specific functions, such that most aspects of networking may be served with a purely file-based API. This will allow programs that use I/O streams or sequential file access, not just network-aware applications, to communicate across networks with standard, consistent file operations. We also implement this paradigm for the Linux operating and evaluate its effectiveness.

In the following section, we describe our proposed file-based API for networking, describing both client and server paradigms, as well as a filename mapping scheme. This is followed by a description of our implementation. Section 4 presents both a qualitative evaluation of our implementation as well as some performance measurements. The paper concludes after a discussion of related work and future directions.

2 File Abstraction for Networking

In order to extend the file abstraction to general-purpose networking, we first must map network-centric functions to a standard file system interface. In particular, we define the steps needed to establish a communication channel using ordinary file operations. As a goal, we wish to ensure no loss of functionality with respect to a sockets-based implementation, at least for the most commonly-used services and features. Furthermore, one should be able to establish a network connection without any special application programming, so any simple file access tool (e.g., `cat`) can be used for network communication.

In this section, we will define a general file-based interface to networking services. As most networking protocols, particularly Internet protocols, involve a server-client model for communication establishment, we take this approach in defining our abstraction. We first look at the simpler of the two, the client side, and then at the somewhat more complex server-side interface. The following discussion primarily centers on TCP, but is applicable more generally, as we will show later.

2.1 General File-based Network Access

As we have discussed earlier, the file abstraction for networking is partially available in UNIX. In particular, read, write, and close operations are available for network streams. Only the channel-establishment operations require the use of specialized, networking functions. To complete the file abstraction for networking, we would like to enable the ordinary `open` call to establish a communication channel.

One method to do this is to follow the paradigm used in UNIX named pipes [3]. We can introduce a new file type, the *named network pipe*. A file of this type is created first. When a program subsequently opens this file, a network connection is established. From then on, the normal file operations can be used for the actual communication. There are a few obvious issues with this approach. First of all, we need a method for specifying the network address of the remote system. This may be specified as options when the file is created, but will require very complex and inelegant modifications to the file-create system call (in contrast, only a simple flag is needed to specify the creation of a pipe). Furthermore, this will introduce yet another special file type to the UNIX environment. Finally, it is not clear how this technique may work for servers that may need to deal with multiple, simultaneous clients.

We take a slightly different approach instead. Rather than using a special file, we use a special virtual file system (VFS). All of the files in this VFS are used for network communication channels, avoiding the need for another new file type. In addition, we can solve the addressing problem by simply mapping the network address space to the file name space in the networking VFS. Thus, for example, a file named `tcp/foo.bar:80`, would be used to communicate to the web server (TCP protocol, port 80) on the machine with the name "foo.bar." The name mapping can also use the IP address of the destination machine as well. These "files" are created on-demand when opened, alleviating the need for a separate file creation step. With this general file paradigm, we now look at the interface details for both client- and server-side operations.

File operations	Socket operation (connection-oriented)	Socket operation (connectionless)
create	socket, addr structure	socket, addr structure
open	connect	bind/connect
read	read	recvfrom/read
write	write	sendto/write
close	close	close
seek	—	—

Table 1: File operations to sockets operation mapping for clients

2.2 Client-side Interface

The client-side interface is summarized in Table 1, for both connection-oriented (i.e., TCP) and connectionless (i.e., UDP) networking, showing the correspondence between file operations and the sockets counterparts. The `read`, `write`, and `close` filesystem operations correspond trivially with the networking counterparts. The file operation `open` has no direct implementation for sockets, but in our paradigm, it corresponds to the establishment of the connection, or binding the receive port for TCP and UDP, respectively. `open` establishes the communication channel associated with the file.

The association between a file and the communication channel parameters is established when the file is created. The name of the file specifies the remote address and port for the communication channel. The actual mapping from name space to address space will be discussed later. Effectively, file creation corresponds to setting up address and sockets data structures with the BSD sockets API. The virtual filesystem performs this create step implicitly on file lookup, so it is not necessary to explicitly create the file in advance, nor to specify the create option to the `open` system call.

With this API, we can now perform operations, like "`echo hello` > *remote_machine:port*," to establish and use a network communication channel. With the implicit file creation on lookup, opening files for input (which does not normally create files), e.g., "`cat` *remote_machine:port*," will also work in this system. To

keep the filesystem uncluttered, an implicit delete on write is used, but garbage collection of unused file entries may still be needed in a practical implementation.

This client-side API provides the most commonly-used Internet protocol client paradigm — use any local port, any interface or IP address to communicate to a specific remote host on a specific port. The actual naming scheme can, however, allow the client to specify a desired local port and address on systems with multiple IP addresses available. More details on the filename mapping is provided in Section 2.4.

File operations	Socket operation (connection-oriented)	Socket operation (connectionless)
create	socket, addr structure	socket, addr structure
open	bind, listen	bind
open	accept	client addr structure
read	(accept), read	recvfrom
write	(accept), write	sendto
close	close	close
seek	—	—

Table 2: File operations to sockets operation mapping for servers

2.3 Server-side Interface

The common-case client networking schema map very nicely to the file interface abstraction. However, for servers, the mapping is a little bit more complex. This is due to the greater range of usage models that are common for Internet servers. However, most functionality can still be made available through a file interface, permitting applications not explicitly designed for networking to be used as server tasks.

The proposed file abstraction for server applications is summarized in Figure 2 as a translation to socket operations. The simple, most commonly-used semantics are optimized in this API. We now discuss in detail how this interface may be used in four different server schema: single-instance and multi-client servers for both connection-oriented and connectionless protocols. These will cover the vast majority of networking service needs.

2.3.1 Simple, Connection-Oriented Server

Here we consider a simple, single-instance TCP server. The server listens on a port, and handles just one client connection. With the filesystem API, this can be implemented very simply. First, the server uses an `open` call to create the file entry and to bind to a port. It then issues a `read` or `write` call to the open file descriptor. This first call is blocking — it performs an implicit wait for the arrival of a client (akin to `accept` with sockets). Subsequent `read` and `write` calls work as usual, transferring data immediately. When the communication session is finished, if the client closes the connection, the server receives an end-of-file indication on read, or a write error. The server can terminate the connection at any time by closing the file descriptor. When the descriptor is closed, the server port is unbound, and made available to other programs.

This very simple mechanism lends itself well to applying file-oriented applications and utilities to do networking, since the entire sequence of system calls is identical to those used for ordinary file access. For example, we can use `cat` *server_port* > *filename* to set up a server that receives data on port *server_port* and saves it to a file named *filename*. A client can simply be `cat` *file* > *server_name:server_port*. With this, we can transfer a file from a client machine to the server machine. When the single file is transferred, the server process (`cp` in this case) receives the end-of-file indicator, and will close the port, and clean up state nicely.

2.3.2 Simple, Connectionless Server

We now look at simple UDP servers. Here, we consider one-way communication, or a simple query-reply transaction. The server binds to a port, reads a datagram, and if necessary, sends back a reply. This can be done with the filesystem API as follows. First, the server program opens a file corresponding to the desired port. This call is equivalent to the `bind` call in the sockets API, and ensures exclusive access to the port by this application. The server then issues a `read` call, which blocks until some data arrives. It can then process this data, and if desired, can send back a message using `write`. The `write` call will send the reply packet to the the client address corresponding to the most recent `read`. This way, explicit addressing operations are not needed for a simple query-response server, and even programs that are not aware of networking can still use the network services correctly. When the server terminates, it issues a `close` call, which unbinds the port, and cleans up any networking state.

As with the simple, connection-oriented semantics, this mechanism works well for file-based utilities. For example, we can implement a simple audio message receiver with `cat` *udp_server_port* > `/dev/audio`. Any datagrams received on UDP port *udp_server_port* will be simply sent to the audio device. Anyone can now send audio messages by using a client `cat /dev/audio` > *server_name:udp_server_port*. Of course, as UDP does not provide error correction like TCP, it is possible to have dropped packets that cause skips in the received audio. Furthermore, if two clients transmit simultaneously, their data may be received interleaved. With connectionless transport protocols, there is no clear indication of

when communications have terminated, so an end-of-file marker is not available to the server. Therefore, it must be stopped manually, or the end of communication must be determined at the application level.

2.3.3 Complex, Connection-Oriented Server

Here, we consider the more general case of a connection-oriented server, where we have a potentially multi-threaded TCP server that can handle multiple clients simultaneously. For this, we do need a server application that is aware of networking, as the semantics of a concurrent server does not make sense with ordinary files. However, the necessary operations can still be handled within the filesystem API. As in the simple case, the server opens a file with a name corresponding to the desired TCP port, setting up the network stack to listen for client connections. Rather than read/write to this open file handle, it now opens the file again. This second `open` is a blocking call, that waits for clients to connect to the server port, and is akin to an explicit `accept` call with sockets. Once a client connection is established, the call returns a file handle that is used to perform all communication with this particular client. The server can call `open` multiple times on the file, and therefore handle multiple, concurrent client connections. Each communication channel can be read, written, and closed independently of the others. When the original file descriptor is closed, the port is unbound, no further clients are allowed to connect, and all states are cleaned up once all of the individual client connections are also closed.

With this mechanism, it is trivial to implement a multithreaded/multiprocess server. The master thread opens the file to bind the port, and then spawns several service threads, each of which also calls `open` and blocks waiting for a client to connect. Once all of the communication is complete, it closes the file and then repeats the process, waiting for more clients. We note that the protections on the server port are kept the same as with a sockets implementation — only the process that initially opened the file corresponding to the port, as well as any child processes forked while holding the descriptor open, are allowed to open the file again and accept client connections.

2.3.4 Complex, Connectionless Server

In this case, too, we consider a server that can handle concurrent clients, but using UDP, rather than TCP. The simple mechanism by which `write` always goes to the client from which we most recently read data, works only for simple query-reply semantics and cannot handle concurrent clients. Instead, we need to be able to send replies to any specific client, even if the most recent read did not correspond to it. As with the connection-oriented mechanism, we can accomplish this using a second `open` call. As before, the server initially opens the file to bind the server port. It can then call `read`, to wait for a datagram to arrive from a client. Now, instead of simply writing a response, we open a one-way channel to the client that sent the message by simply opening the file a second time. This second file handle can be used to send replies to the particular client, even if messages from other clients have been read. As there is no concept of a connection, all incoming messages will still be received by reading the original file descriptor, but replies may be deferred and sent back at any time using the subsequently opened file descriptors.

With this, we can easily implement a UDP query-reply server that can handle concurrent clients, which may be needed if the replies are very long, or if the queries involve long processing times. The main thread opens the file to bind the port and reads any query messages that arrive. For each message, it opens the file again to establish the response channel to the client, and spawns a thread to handle the query and reply. It can then receive new client queries and create concurrent threads to handle them. One limitation with this is that any subsequent communication from an existing client will be read through the original file handle, so it will be treated as a new query unless the application is properly designed to handle this case. In the following section, we propose a mechanism to determine the client from which each received datagram originates.

2.3.5 Explicit Client Addresses

An important consideration in all of the server operations described so far is the implicit addressing of replies to the clients. This allows the use of network-unaware programs to perform network communication with the proposed file interface. However, at times, it may be necessary for an application to know the actual client address and source ports. In particular, with simultaneous UDP clients connected to a server, the source network address and port are useful in sorting the received datagrams by the client.

The network address is not actually a part of the communication stream from the client, so it should be considered metadata or "out-of-band" data, and should not be directly available through `read` and `write` calls. However, for network-aware applications, we can request the address information in-band by setting an option flag on the file handle. This may be set using an `ioctl` call on the open server file handle, but additionally, in keeping with our file abstraction, can be requested implicitly with the original `open` call by appending the "!" symbol to the filename corresponding to the desired server port.

With the explicit address option is set, the first read from a new TCP connection will provide the address and

Filename format	Type	Local IP	Local port	Remote IP	Remote port
port[!]	server	any	*port*	—	—
addr:*port*	client	any	any	*addr*	*port*
addr:*port*::[!]	server	*addr*	*port*	—	—
lport:*addr*:*port*	client	any	*lport*	*addr*	*port*
laddr:*lport*:*addr*:*port*	client	*laddr*	*lport*	*addr*	*port*

Table 3: Filename interpretation for TCP and UDP — the addresses may be specified numerically in dotted-decimal format, or as symbolically as hostnames; a server filename with "!" symbol appended will use the explicit client addressing option.

port information of the client, while subsequent reads will provide received data as usual. Writes are not affected. With UDP, each datagram read will be prepended with the client address and port, as well as the length of the datagram. The length is useful in case multiple `read` calls are required for any particular datagram. For UDP, `write` calls are affected as well. Rather than implicitly sending a reply to the last datagram received, each write through the original server file handle must be prepended with the client address and port, as well as the datagram length. If the file corresponding to the server port is reopened (as in the complex, connectionless server scenario above), a write-only channel to a specific client is established, so the writes to these secondary file handles are not affected. However, now, the secondary `open` calls use a destination address corresponding to the last communication (read or write) on the original file handle.

We note that with the explicit addressing option, a UDP peer-to-peer paradigm is made possible. Here, all hosts are servers, listening for messages on a particular port, as well as clients, sending messages from the port to remote peers. Under the file-based interface, once the file corresponding to the desired server port is opened, we can trivially initiate transmission to any remote machine through the same file handle with explicit addressing enabled. Hence, with the addition of this mechanism of providing explicit client addresses, the file abstraction for networking becomes sufficiently general to handle almost all common networking scenarios through a consistent, file-based interface.

2.4 Name and Address Space Mapping

Having specified the behavior of the file interface to the networking subsystem, we complete our design with a mapping of the network address space to the filesystem namespace. In order to keep real files separate from the networking namespace, we ensure that the networking virtual files are accessed through a separate virtual filesystem that is mounted somewhere on the system (assume `/net` as the mountpoint). Within this filesystem, we need to differentiate between different types of networking protocols, as this may affect the address mapping as well as the file operations behavior. Currently, we have only considered TCP and UDP Internet protocols, although we can envision a lower-level raw-IP or higher-level HTTP protocol also available. Each protocol is provided a separate subdirectory in the root virtual filesystem (e.g., `/net/tcp/` or `/net/udp/`). File entries under these protocol directories correspond to communication channels, and are created on-demand as explained earlier. The names of these files incorporate addressing information needed to establish and use a network communication channel.

To completely specify a communication channel in TCP or UDP, we need a 4-tuple: Host1 IP address, Host1 port, Host2 IP address, and Host2 port. We can map this directly to a filename by simply concatenating the elements, expressed as strings, with a field delimiter (e.g., `Host1_address:Host1_port:Host2_address:Host2_port`). However, in general, when establishing a communication channel, we need not fully specify all of these elements – many can be implicitly set. For example, when a client needs to connect to some server, it only needs to specify the remote address and port, since the local IP address can be automatically supplied, and any available local port may be used. Additionally, a server application typically only needs to specify the port on which it will provide service. Hence, our mapping scheme accepts a variable number (between 1 and 4) of addressing elements, expressed as strings and concatenated using a ":" delimiter, as the name of the file corresponding to these communication parameters. The number of elements and position of the delimiter indicates the type of networking service desired, as a server or client, as well as the addresses and ports to be used, as summarized in Table 3. This mapping allows great flexibility, even allowing applications to bind to a particular local IP address multiple are available. We note that the elements of the name can be numerical or symbolic. Numerical addresses are specified in dotted-decimal format (e.g., 10.0.0.9). Anything else is considered a hostname, and an implicit DNS query will be performed on behalf of the application that tries to access the file. Ports must be specified as base 10

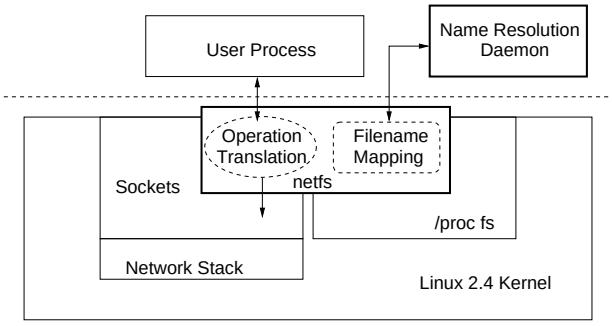

Figure 1: Software architecture for the implementation of networking through a file abstraction

numbers, or as well-known port aliases, (e.g., ftp or http). Finally, for the server-class mappings, we can append a "!" to the filename to indicate explicit client addressing, as detailed in Section 2.3.5.

3 Implementation

We have built a working prototype of the file-mapped networking system on top of the Linux operating system. It implements most of the API specified above, but is currently limited to the TCP protocol. Figure 1 shows the major components of the overall system. The core of our file abstraction is implemented as a loadable kernel module, called `netfs`, that extends the Linux 2.4 kernel. For convenience, it makes use of the kernel's built-in `/proc` filesystem, to avoid re-implementing many of the basic functions needed for a virtual file system.

User processes make use of the networking services through filesystem operations. These are passed by the `/proc` filesystem to our `netfs` module. These are then translated by our module according to the specifications outlined earlier into the equivalent networking functions, which are called via an internal hook into the sockets subsystem. In particular, `read`, `write`, and `close` operations are passed through with little additional processing. Much of the work in the `netfs` module is in appropriately handling the `open` call, setting up networking structures and file handles as needed.

A major subcomponent of the module handles file lookup in the virtual filesystem. This part performs the implicit creation of files on lookup, so that even if `open` is called without a create option, a new network connection can be established. This is particularly useful when redirecting standard input, as a read-only open typically would not create a nonexistent file. Along with lookup, this part of the module also interprets the filename according to the discussion in Section 2.4, and creates internal state to represent the desired network addresses and communication parameters.

As a network host may be specified by name, the module needs to be capable of querying DNS service to resolve the actual network address of a host. Rather than performing this complex activity from within the kernel, the module hands off the actual name resolution to a user-level daemon that uses the standard library routines to determine host addresses. Care is taken to ensure that this name resolution interface can support concurrent lookups, and that one slow DNS query will not force other name lookups/connection attempts to block. This architecture also provides the option of adding a name cache to the user-level daemon, greatly reducing the overheads of repeated queries. One limitation of this approach is that additional latencies are incurred due to multiple context switches when a connection requiring name resolution is initiated. The performance of this implementation is evaluated in the following section.

One issue with the implementation is that the internal virtual file interface separates the file open function from the filename lookup. As a result, the implicit file creation actually occurs on all file lookups, not just in conjunction with the `open` call. As a result, executing `ls /proc/netfs/tcp/foo.bar:80` will create the file entry, ready to be opened as a new connection. This explicit lookup may result in unused file entries, cluttering the virtual file system. In addition, orphaned entries may result from bad crash behavior, if a process fails to complete the `open` call after having performed the file lookup. To alleviate this, we need to implement a mechanism of periodically garbage collecting unused file entries. This will be incorporated into a future implementation of our file-based networking interface.

4 Evaluation

Using our implementation of a file abstraction for general-purpose networking, we seek to evaluate the performance and utility of this mechanism. Our evaluation can be divided into three parts. First, we look at the completeness of this implementation with respect to a sockets-based interface. Next, we try to evaluate the ease of use with the new networking scheme. Finally, we benchmark the system to determine performance costs in using our networking interface implementation.

4.1 Completeness

We have designed our file-based networking API such that it captures the behavior and functionality needed for a vast majority of networking applications. However, in the process, we have tuned the interface for the most common usage patterns, and may be missing some niche features

```
char buf[1024];
int fd;
struct sockaddr_in addr;
struct hostent* host;
fd = socket (AF_INET, SOCK_STREAM, 0);
bzero ((char*) &addr, sizeof(addr) );
host = gethostbyname("foo.bar");
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = *((long*)(host->h_addr));
addr.sin_port = htons(8000);
connect (fd, (struct sockaddr *) &addr,
        sizeof (addr) );
read(fd, buf, 1024);
close(fd);
```

(a)

```
char buf[1024];
int fd;

fd=open("/proc/netfs/tcp/foo.bar:8000",O_RDONLY);

read(fd, buf, 1024);
close(fd);
```

(b)

Figure 2: Client code to read 1KB from a server: (a) Sockets implementation; (b) With a file-based interface

available through the traditional network programming interface.

Our file interface consolidates multiple networking functions into a single file `open` call. As a result, the individual networking functions are not directly available. Through our interface, it is not possible to only bind to a TCP port — the `open` forces the equivalent of `listen` or `connect` to also be called.

In addition, our interface implicitly performs name resolution on behalf of the process initiating communication. However, the results of the DNS query, i.e., the host address(es), are not directly available to the process. We do not provide a function to explicitly perform a DNS query. Instead, applications should rely on existing library support for this service (e.g., `gethostbyname` function). We note that the library services may be ported to use our interface, rather than sockets, when communicating to remote DNS servers.

Finally, we do not currently support "out-of-band" or "urgent" messages over the communication stream. This is a feature available in TCP [8] to allow high-priority data to be sent immediately, ahead of bulk transfers that may be buffered for the connection. This does not map nicely to file semantics, and requires a more complex interface. We can implement out-of-band communication through `ioctl` functions on the open connection file descriptor. However, as `ioctl` functions tend to be OS-specific, this may limit the portability of this interface across operating systems. We leave the specification of out-of-band communication for future work.

4.2 Ease of Use

One important goal and characteristic of the file-based interface is that it is very easy to use. Although this is a very subjective property, and not readily quantifiable, we can show examples of how it can very readily be used in programs, reducing total coding necessary for networking.

We first look at the typical schema for a client program that initiates a TCP connection to a remote server and reads up to 1024 bytes over the channel. Figure 2(a) shows the code typically needed for a client written in C using sockets interface, and standard library routines to handle server name resolution. This code involves two complex data structures, a couple of magic constants, several function calls, and a few type casts to compile properly. Yet, this is the canonical client code, very similar to textbook examples [11], but with error handling removed. In contrast, Figure 2(b) shows the equivalent client code, but using file-based networking interface. Here, no complex data structures are needed, and only a single call is needed to establish the connection. Even if error handling were added, only the return value of the `open` needs to be checked to detect problems and ensure proper program behavior.

In addition to simplifying the code written to implement clients, the file abstraction can greatly simplify server implementations as well. Figure 3 shows a shell script that implements a simple chat server that can handle multiple, concurrent clients. Most of the code (function `PROG`) implements a single instance of the the interactive chat service for a single user, assuming standard input and output communicate to this user. The `chatlog.txt` file is used both to log all of the messages received and as an implicit IPC mechanism, as each instance will output all

```
#!/bin/bash
CHATFILE=chatlog.txt
PORT=8000

function INIT() { touch $CHATFILE; }
function PROG() {
        echo -n "Username: "
        read username
        echo $username joined >> $CHATFILE
        tail -f $CHATFILE &
        while read nextline
                do echo $username ": " $nextline >> $CHATFILE; done
        echo $username left >> $CHATFILE;
}

# General multiclient TCP server
SERVERFILE=/proc/netfs/tcp/$PORT
INIT
(while ( echo -n ) do                                   # Loop forever
        (PROG &) <> $SERVERFILE 1>&0 2>&0               # Wait for client connection
                                                        # and redirect stdin, stdout, stderr
done) 3< $SERVERFILE                                    # Open original file handle
                                                        # to bind and listen on server port
```

Figure 3: Chat server supporting multiple concurrent clients, written as a shell script

new messages appended to this file through the `tail -f` background process. The actual networking part of this server is confined to just the last few lines. The script opens the file corresponding to the desired port (binding the port and establishing the listen queue), and then executes an infinite loop that launches instances of the chat code in the background. For each of these instances, the server file is reopened, which blocks until a client connects, and the standard input, output, and error are redirected to this communication channel handle.

This example shows how little code really is necessary to implement sophisticated networking services. Using the file abstraction will require less time in writing code for the actual networking part of programs, and may help reduce the likelihood of bugs due to programmer error. This example also illustrates how the abstraction greatly enhances the power of the existing, network-unaware tools, allowing the use of simple programs, such as `tail`, and shell scripts to implement concurrent, multi-user Internet services.

4.3 Performance

The previous section describes the merits of the file-based API for networking. In this section, we evaluate the performance overheads incurred with our Linux implementation. The test system consists of a uniprocessor PC, with a 2.0 GHz Intel Pentium processor. The processor incorporates HyperThreading technology, which allows 2 concurrent process states to be active and makes the processor appear as a dual processor core. However, for testing, we disable this feature, and use a uniprocessor OS kernel. The system runs Linux 2.4.20, as shipped on a vanilla RedHat 9.0 distribution, with only our module for file-based networking added. The platform also has a 100 Mbps Ethernet adapter.

We developed several small microbenchmarks to test the operation overheads when using our modified API, as compared to the traditional networking interface. All timings are based on the processor timestamp register, a 64-bit counter that simply counts the processor clock cycles. As this counter operates at full core frequency, the timestamp values permit very high-resolution time measurements, on the order of nanoseconds. This register is accessed through a single machine instruction, `rdtsc`, so little overhead is incurred in taking the measurements.

4.3.1 Connection-Establishment Delay

As our interface primarily affects only the connection-establishment path for networking, we first measure the additional latencies involved in this step. We set up two

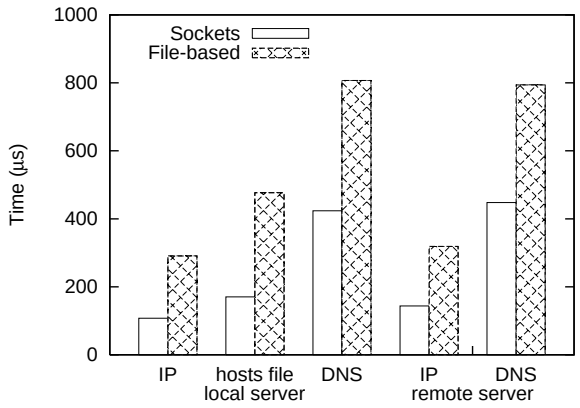

Figure 4: Connection establishment time for local and remote servers

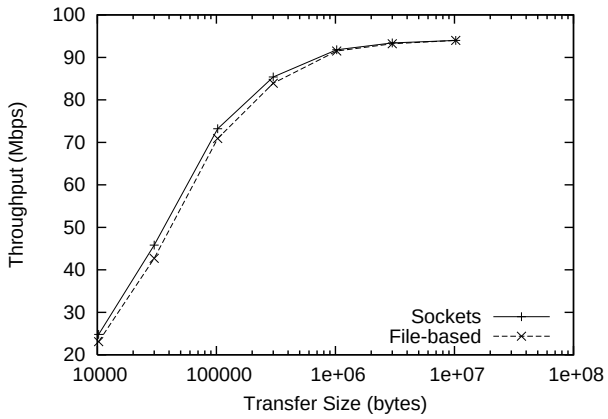

Figure 5: Throughput for sockets- and file-based networking

versions of a client process, one using our networking API and another using sockets, on the test machine. We time how long TCP connection establishment to a locally running server takes, in cases where the server is identified by address, by a name in the local `/etc/hosts` file, and by a name that requires DNS lookup. We note that the DNS server is located on the same Ethernet segment and is the primary server for all of the host names we use. We also repeat the DNS and IP based experiments with a remote server on the same Ethernet segment.

Figure 4 shows the time required to establish a TCP connection under these conditions, averaged over 100 repetitions of each experiment. Generally, the `open` call through our virtual filesystem does incur greater overhead than the direct socket calls. Part of this is implementation related — our prototype basically translates our API and internally makes use of the same network functions as do the socket calls, so we cannot do any better than the sockets implementation. The other source of overhead is inherent to the design: the `open` call involves string processing for file lookup, multiple levels of directory traversal, and then further processing to interpret the name and extract addresses in our interface. When connecting by host name rather than IP address, overheads are increased further, due to multiple context switches incurred when the user-level daemon is invoked to perform name resolution.

We note, however, that if the client program uses the traditional sockets interface and standard library functions to resolve hostnames, there is an additional overhead associated with initializing the resolution library. In particular, the first call of `gethostbyname` by any particular process requires opening several configuration files, e.g. `/etc/nsswitch.conf`, `/etc/hosts`, etc., and will incur approximately 700 μs longer than normal. This is not a problem with our implementation, as the name lookup daemon is long-lived and initializes this only once, so the penalty is not inflicted on future processes. This means a latency-sensitive client that makes a single connection by host name will take significantly longer with a traditional sockets-based approach than with our file-based API.

We also evaluate the time required for a server to accept the next incoming client, through the sockets interface as well as our file-based approach. We ensure that a client is waiting on the listen queue so the server does not block, and then measure the time required to execute the system calls to get a handle to this client. Here, we see the file-based interface in the worst light, as the sockets `accept` call is significantly faster than the `open` call. With sockets, this function requires just 10.7 μs, and is one of the fastest network operations if blocking does not occur. With the overheads of the `open` call and the required directory traversal and string manipulation, this time increases to 114.4 μs with the file-based API. We believe an optimized implementation that takes advantage of the kernel filesystem directory caches aggressively and also directly calls functions in the network stack can improve this result in the future.

4.3.2 Data Transfer Overheads

In addition to the connection-establishment times, we also investigate overheads for actual data transfer operations. By instrumenting the client program, we measure the time required for a `read` or `write` call to execute. We use short transfers (16 bytes), and ensure that no blocking occurs (i.e., ensure data is available before a read operation). When using the sockets interface, these operations take an average of 8.9 μs. For our file-based technique, this increases slightly to 12.2 μs. We note that these measurements only account for the processing overheads to move

data to the networking stack, and does not include the actual network data transfer times.

Of course, individual data operation execution times are not usually very important to most applications. Rather, the total time to transfer data, or the sustained throughput rate is generally a more useful metric of performance. We measure the total time required to transfer varying size blocks of data across a fast Ethernet link using both the traditional networking API and our file-based interface. The results are summarized in Figure 5. The achieved transfer rate, which includes connection establishment and shutdown times, is plotted against the total bytes transfered over the connection. For larger transfers, the overheads are amortized, and the achieved throughput for both networking API's converge to the same, network-limited value. For small transfers, the connection-establishment and data operation overheads are more significant, and the traditional sockets-based system can provide a slightly higher throughput. However, across this spectrum, our file-based mechanism stays within approximately 7% of throughput of sockets-based implementation.

5 Related Work

Very little work has been done in fully extending the UNIX file abstraction to general-purpose networking. This may be due to the prevalence of Berkeley sockets [2] and compatible interfaces, and a lack of motivation to break with the established standard.

One major effort to extend file abstractions has been undertaken in the Plan 9 operating system [7]. One of the goals of Plan 9 is to push the file abstraction as far as possible and use it for all services, including general purpose networking [9]. However, the file interface in provided is far different from the abstraction we would like. Plan 9 networking services provide a set of files, including one for the control path. Most operations, such as setting up a channel, are done by writing into the control-path file, and then data operations read and write from a separate file representing the data path. Applications still need to be network-aware with this interface, unlike our abstraction.

Other work [5] uses a filesystem abstraction to access files over the network through the HTTP protocol. This involves a mapping of filenames to network addresses and remote files. However, this does not provide general-purpose networking, but rather focuses on HTTP, and using it to implement a remote filesystem.

Some efforts have developed mechanisms by which non-network-aware applications may communicate to remote services. One such mechanisms is a tool called `netcat` [4], which makes network connections, and redirects standard input and output to the remote host. This makes it possible to obtain some of the benefits of our file-system abstraction for networking. In particular, we can use any tool that relies on standard input and output, piped through `netcat`, to operate over network connections. However, there are limitations with the `netcat` approach. Using `netcat` as a server provides only limited functionality. In particular, one cannot have a concurrent / multiclient / multithreaded servers. Also, since `netcat` relies on redirecting standard input and output, it is awkward / difficult to use in an application with multiple network connections. This would require spawning multiple `netcat` processes, each with a dedicated pair of UNIX pipes, or other IPC channels, to communicate with the main application. In contrast, all of these features are available directly through the `open` call in our file-abstraction, without the hassles of multiple interacting processes.

Recent versions of the GNU Bourne-Again Shell (`bash`) [10] provide a limited version of networking through a file abstraction. The shell interprets file names of the form `/dev/tcp/host/port` and `/dev/udp/host/port` as network channels to remote hosts, using TCP and UDP, respectively. This permits us to redirect standard input or output to network communication channels. The `bash` feature is limited to acting as a client, and one cannot set up a server to listen on a port. Furthermore, this abstraction is interpreted by the shell, and it cannot be used internally by a program that directly tries to open the file.

Although not directly related to general-purpose networking, several recent papers [1, 6, 12] devise methods of extending an OS and implementing new filesystem abstractions. Two of these [1, 6] devise techniques of implementing filesystems at the user-level, by intercepting system calls or through a user-level NFS loopback server. Either of these mechanisms could have been used to implement our file abstraction for networking, but would have involved greater overheads due to context switching between processes on each network operation. The third paper [12] also devises techniques to simplify the creation of new file-abstractions, but at a kernel-level. By using a generic wrapper filesystem that provides all of the functions needed to interact with the kernel, only a few functions need to be written in order to correctly implement a new virtual file system. Our implementation follows a similar technique, but relies on Linux's `/proc` filesystem to provide many of the generic functions required for a working virtual file system implementation.

6 Conclusions and Future Work

In this paper, we have argued for a general extension file abstractions to general-purpose networking. We have proposed mappings of file operations and file names to network functions and addresses that permit most client and server schema to be expressed through the file abstraction. This powerfully extends existing applications, allowing programs that are not network aware to access and use remote services through ordinary file operations. We have shown through implementation and examples how total code necessary to implement complex Internet server behavior can be greatly simplified with a file-based interface to networking services. These benefits are available at a very low cost — even in our prototype implementation, connection establishment time is increased by no more than 400 μs, and total throughput reduced by less than 7% relative to a traditional networking API.

This work has demonstrated the usefulness of a file-based networking API, and shown how to cover the most common schema for network communication. In the future, we wish to achieve more thorough coverage of less common paradigms that can be supported under the traditional sockets API. In particular, we would like to ensure a clean abstraction for out-of-band messages in TCP. With the advent of IPv6, we would like to extend the filename mapping mechanisms to ensure compatibility with common addressing modes employed in IPv6 networks. Finally, file semantics provide some access paradigms that are not currently available to networking. In particular, concurrent reads and writes to the same instance of a file is possible, as well as several methods of file locking and access control. We would like to investigate how such file and filesystem notions may be applicable to networking, and whether new functionality may be provided to networking applications.

References

[1] ALEXANDROV, A., IBEL, M., SCHAUSER, K., AND SCHEIMAN, C. Extending the operating system at the user level: the ufo global file system. In *Proceedings of 1997 USENIX Annual Technical Conference* (Jan. 1997).

[2] BERKELEY. *Unix Programmer's Manual: 4.2 Berkeley Software Distribution*. 1983.

[3] DOLOTTA, T. A., OLSSON, S. B., AND PETRUCELLI, A. G. *Unix User's Manual*. Bell Laboratories, 1980.

[4] GIACOBBI, G. *The GNU Netcat – Official homepage*. http://netcat.sourceforge.net/.

[5] KISELYOV, O. A network file system over http: Remote access and modification of files and files. In *1999 USENIX Annual Technical Conference, FREENIX track* (June 1999).

[6] MAZIERES, D. A toolkit for user-level file systems. In *Proceedings of 2001 USENIX Annual Technical Conference* (June 2001).

[7] PIKE, R., PRESOTTO, D., DORWARD, S., FLANDRENA, B., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. Plan 9 from Bell Labs. *Computing Systems 8*, 3 (1995), 221–254.

[8] POSTEL, J. Transmission Control Protocol. RFC 793, Sept. 1981.

[9] PRESOTTO, D., AND WINTERBOTTOM, P. The organization of networks in Plan 9. In *Proc. of the Winter 1993 USENIX Conf.* (Jan. 1993), pp. 271–280.

[10] RAMEY, C. *The GNU Bourne-Again SHell*. http://cnswww.cns.cwru.edu/ chet/bash/bashtop.html.

[11] STEVENS, W. R. *UNIX Network Programming*. Prentice Hall, 1990.

[12] ZADOK, E., BADULESCU, I., AND SHENDER, A. Extending file systems using stackable templates. In *Proceedings of 1999 USENIX Annual Technical Conference* (June 1999).