# Using TCP through sockets

David Mazières (mostly),
Frank Dabek, Eric Peterson,
and Thomer M. Gil

## 1 Introduction

This document provides an introduction to using sockets on Unix systems. We first cover the basics of Unix file I/O and then discuss the use of sockets to communicate over a network using TCP.

Readers of this document are assumed to be proficient in the C and C++ programming languages. Also, you should be comfortable compiling programs, reading manual pages, etc., on a Unix operating system. Detailed knowledge of network standards or Unix I/O is not required.

Noting the following typesetting conventions may assist the reader: words set in sans serif type (such as open) are Unix system calls; any text set in a mono spaced font (such as `nbytes`) is either a C/C++ code fragment, a C/C++ object or name, or a Unix command.

## 2 File descriptors

Most I/O (input/output) on Unix systems takes place through the read and write system calls[1]. Before discussing network I/O, it helps to understand how these functions work even on simple files. If you are already familiar with file descriptors and the read and write system calls, you can skip to the next section.

All read and write operations must be performed on file descriptors, non-negative integers which are created via a call to system calls such as open for files, or socket and accept for sockets. File descriptors remain bound to files/sockets even when they are renamed, deleted, or undergo permission changes that revoke access[2].

By convention, file descriptor numbers 0, 1, and 2 correspond to standard input, standard output, and standard error respectively. Thus a call to printf will result in a write to file descriptor 1.

Section 2.1 shows a very simple program that prints the contents of files to standard output—just like the UNIX cat command. The function `typefile` uses four system calls to copy the contents of a file to the standard output.

[1] High-level I/O functions such as fread and fprintf are implemented in terms of read and write.

[2] Note that not all network file systems properly implement these semantics.

- `int open(char *path, int flags, ...);`

  The open system call requests access to a particular file. `path` specifies the name of the file to access; `flags` determines the type of access being requested—in the case of this example read-only access. open ensures that the named file exists (or can be created, depending on `flags`) and checks that the invoking user has sufficient permission for the mode of access.

  If successful, open returns a file descriptor.

  If unsuccessful, open returns −1 and sets the global variable errno to indicate the nature of the error. The routine perror will print "filename: error message" to the standard error based on errno.

- `int read (int fd, void *buf, int nbytes);`

  read will, by default, block until there is data to read from `fd` and then reads that data up to `nbytes` bytes of data into memory starting at `buf`. (However, read will return immediately in case of an end of file or in case of an error.) Upon success, read returns the number of bytes actually read, which may very well be less than `nbytes`. The case in which read returns fewer than `nbytes` is often called a "short read" and is a common source of errors. If read returns 0, this indicates an end of file. If it returns −1, this indicates an error.

- `int write (int fd, void *buf, int nbytes);`

  write will write up to `nbytes` bytes of data at `buf` to file descriptor `fd` and returns the number of bytes actually written, which equals `nbytes`[3]. Write returns −1 to indicate an error.

- `int close (int fd);`

  close deallocates a file descriptor. Systems typically limit each process to 64 file descriptors by default (though the limit can sometimes be raised substantially with the setrlimit system call). Thus, it is a good idea to close file descriptors after their last use so as to prevent "too many open files" errors.

## 2.1 `type.c`: Copy file to standard output

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdlib.h>

void
typefile (char *filename)
{
  int fd, nread;
  char buf[1024];
```

[3]Unfortunately, the return value may be less than `nbytes` if the file descriptor is non-blocking, but ignore that for now.

```
  fd = open (filename, O_RDONLY);
  if (fd == -1) {
    perror (filename);
    return;
  }

  while ((nread = read (fd, buf, sizeof (buf))) > 0)
    if(write (1, buf, nread) < 0)
      perror("write");

  close (fd);
}

int
main (int argc, char **argv)
{
  int argno;
  for (argno = 1; argno < argc; argno++)
    typefile (argv[argno]);
  exit (0);
}
```

# 3 TCP/IP Connections

## 3.1 Introduction

TCP is the reliable protocol many applications use to communicate over the Internet. TCP provides a stream abstraction: two processes, possibly on different machines, each have a file descriptor. Data written to either descriptor will be returned by a read from the other. Such network file descriptors are called sockets in Unix.

Every machine on the Internet has a unique, 32-bit IP (Internet protocol) address[4]. An IP address is sufficient to route network packets to a machine from anywhere on the Internet. However, since multiple applications can use TCP simultaneously on the same machine, another level of addressing is needed to disambiguate which process and file descriptor incoming TCP packets correspond to. For this reason, each end of a TCP connection is named by 16-bit port number in addition to its 32-bit IP address.

So how does a TCP connection get set up? Typically, a server will listen for connections on an IP address and port number. Clients can then allocate their own ports and connect to that server. Servers usually listen on well-known ports. For instance, finger servers listen on port 79, web servers on port 80 and mail servers on port 25. A list of well-known port numbers can be found in the file `/etc/services` on any Unix machine.

The Unix `telnet` utility will allow to you connect to TCP servers and interact with them. By default, `telnet` connects to port 23 and speaks to a `telnet` daemon that runs login. However, you can specify a different port number. As mentioned above, on many machines port 25 runs a mail server:

[4]This discussion will assume IP version 4. Version 6 of IP significantly expands the address space

```
athena% telnet PACIFIC-CARRIER-ANNEX.mit.edu 25
FIXTHIS
quit
quit
^]
telnet> q
Connection closed.
athena%
```

Note that in order to quit telnet, you must type Control-] followed by q and return.

As another example, let's look at the finger protocol, one of the simplest widely used TCP protocols. The Unix `finger` command takes a single argument of the form *user@host*. It then connects to port 79 of *host*, writes the *user* string and a carriage-return line-feed over the connection, and dumps whatever the server sends back to the standard output. We can simulate the `finger` command using `telnet`. For instance, using `telnet` to do the equivalent of the command `finger help@mit.edu`, we get:

```
athena% telnet mit.edu 79
...including Athena's default telnet options: "-ax"
Trying 18.72.0.100...
Connected to mit.edu.
Escape character is '^]'.
help
These help topics are available:

about           general         options         restrictions    url
change-info     motd            policy          services        wildcards

To view one of these topics, enter "help name-of-topic-you-want".
...
Connection closed by foreign host.
athena%
```

## 3.2 TCP client programming

Now let's see how to make use of sockets in C. Section 3.3 shows the source code to a simple finger client that does the equivalent of the last `telnet` example of the previous section. In general, a client wishing to create a TCP connection to a server first calls socket to create a socket, optionally calls bind to specify a local address, and finally connects to the server using the connect system call.

The function `tcpconnect` in `myfinger.c`, Section 3.3, shows all the steps necessary to connect to a TCP server. It makes the following system calls:

- `int socket (int domain, int type, int protocol);`

The socket system call creates a new socket, just as open creates a new file descriptor. socket returns a non-negative file descriptor number on success, or $-1$ on an error.

When creating a TCP socket, `domain` should be `AF_INET`, signifying an IP socket, and `type` should be `SOCK_STREAM`, signifying a reliable stream. Since the reliable stream protocol for IP is TCP, the first two arguments already effectively specify TCP. Thus, the third argument can be left 0, letting the operating system assign a default protocol (which will be `IPPROTO_TCP`).

Unlike file descriptors returned by open, you can't immediately read and write data to a socket returned by socket. You must first assign the socket a local IP address and port number, and in the case of TCP you need to connect the other end of the socket to a remote machine. The bind and connect system calls accomplish these tasks.

- `int bind (int s, struct sockaddr *addr, int addrlen);`

  bind sets the local address and port number of a socket. `s` is the file descriptor number of a socket. For IP sockets, `addr` must be a structure of type `sockaddr_in`, usually defined as follows. Parameter `addrlen` must be the size of `struct sockaddr_in` (or whichever structure one is using).

```
struct in_addr {
   u_int32_t s_addr;
};

struct sockaddr_in {
   short sin_family;
   u_short sin_port;
   struct in_addr sin_addr;
   char sin_zero[8];
};
```

  You shouldn't define these structures yourself. Rather, you should include `<netinit/in.h>`. Different versions of Unix may have slightly different structures. However, all will have the fields `sin_family`, `sin_port`, and `sin_addr`. All other fields should be set to zero. Thus, before using a `struct sockaddr_in`, you must call `bzero` on it, as is done in `tcpconnect`. Once a `struct sockaddr_in` has been zeroed, the `sin_family` field must be set to the value `AF_INET` to indicate that this is indeed a `sockaddr_in`. (Bind cannot take this for granted, as its argument is a more generic `struct sockaddr *`.)

  `sin_port` specifies which 16-bit port number to use. It is given in network (big-endian) byte order[5], and so must be converted from host to network byte order with `htons`.

[5] To avoid ambiguity, the socket standard specifies a 'network' byte order (which happens to be big-endian). The calls htons and ntohs convert shorts back and forth between byte orders. These calls are macros and on big-endian machines have no effect. To ensure portability, however, they should always be used.

It is often the case when writing a TCP client that one wants a port number but doesn't care which one. Specifying a `sin_port` value of 0 tells the OS to choose the port number. The operating system will select an unused port number between 1,024 and 5,000 for the application. Note that only the super-user can bind port numbers under 1,024. Many system services such as mail servers listen for connections on well-known port numbers below 1,024. Allowing ordinary users to bind these ports would potentially also allow them to do things like intercept mail with their own rogue mail servers.

`sin_addr` contains a 32-bit IP address for the local end of a socket. The special value `INADDR_ANY` tells the operating system to choose the IP address. This is usually what one wants when binding a socket, since one typically does not care about the IP address of the machine on which it is running.

- `int connect (int s, struct sockaddr *addr, int addrlen);`

  connect specifies the address of the remote end of a socket and connects to it. The arguments are the same as for bind, with the exception that one cannot specify a port number of 0 or an IP address of `INADDR_ANY`. Connect returns 0 on success or $-1$ on failure.

  Note that one can call connect on a TCP socket without first calling bind. In that case, connect will assign the socket a local address as if the socket had been bound to port number 0 and address `INADDR_ANY`. The example finger calls bind for illustrative purposes only.

The above three system calls create a connected TCP socket, over which the finger program writes the name of the user being fingered and reads the response. Most of the rest of the code should be straight-forward, except you might wish to note the use of `gethostbyname` to translate a hostname into a 32-bit IP address.

The following C/C++ code, like all examples in this document, can be fed to a C++ compiler to yield an executable.

## 3.3 `myfinger.c`: A simple network finger client

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define FINGER_PORT 79
#define bzero(ptr, size) memset (ptr, 0, size)

/* Create a TCP connection to host and port.  Returns a file
 * descriptor on success, -1 on error. */
```

```c
int
tcpconnect (char *host, int port)
{
  struct hostent *h;
  struct sockaddr_in sa;
  int s;

  /* Get the address of the host at which to finger from the
   * hostname.  */
  h = gethostbyname (host);
  if (!h || h->h_length != sizeof (struct in_addr)) {
    fprintf (stderr, "%s: no such host\n", host);
    return -1;
  }

  /* Create a TCP socket. */
  s = socket (AF_INET, SOCK_STREAM, 0);

  /* Use bind to set an address and port number for our end of the
   * finger TCP connection. */
  bzero (&sa, sizeof (sa));
  sa.sin_family = AF_INET;
  sa.sin_port = htons (0);                  /* tells OS to choose a port */
  sa.sin_addr.s_addr = htonl (INADDR_ANY);  /* tells OS to choose IP addr */
  if (bind (s, (struct sockaddr *) &sa, sizeof (sa)) < 0) {
    perror ("bind");
    close (s);
    return -1;
  }

  /* Now use h to set set the destination address. */
  sa.sin_port = htons (port);
  sa.sin_addr = *(struct in_addr *) h->h_addr;

  /* And connect to the server */
  if (connect (s, (struct sockaddr *) &sa, sizeof (sa)) < 0) {
    perror (host);
    close (s);
    return -1;
  }

  return s;
}

int
main (int argc, char **argv)
{
  char *user;
  char *host;
  int s;
  int nread;
  char buf[1024];
```

```
  /* Get the name of the host at which to finger from the end of the
   * command line argument. */
  if (argc == 2) {
    user = malloc (1 + strlen (argv[1]));
    if (!user) {
      fprintf (stderr, "out of memory\n");
      exit (1);
    }
    strcpy (user, argv[1]);
    host = strrchr (user, '@');
  }
  else
    user = host = NULL;
  if (!host) {
    fprintf (stderr, "usage: %s user@host\n", argv[0]);
    exit (1);
  }
  *host++ = '\0';

  /* Try connecting to the host. */
  s = tcpconnect (host, FINGER_PORT);
  if (s < 0)
    exit (1);

  /* Send the username to finger */
  if (write (s, user, strlen (user)) < 0
      || write (s, "\r\n", 2) < 0) {
    perror (host);
    exit (1);
  }

  /* Now copy the result of the finger command to stdout. */
  while ((nread = read (s, buf, sizeof (buf))) > 0)
    write (1, buf, nread);

  exit (0);
}
```

## 3.4 TCP server programming

Now let's look at what happens in a TCP server. A TCP server, like a client, begins by calling socket to create a socket and by binding the socket to a well-known port using bind (although optional for clients, servers nearly always call bind to specify the port on which they will operate). Following the bind operation, server and client paths diverge: instead of connecting the socket, a server will instead call listen followed by accept. These functions, described below, alert the operating system to accept new connects and, for each connection, create a new, connected socket which will be returned by accept.

Section 3.6 shows the complete source code to a simple finger server. It listens for clients on the finger port, 79. Then, for each connection established, it reads a line of data, interprets it as the name of a user to finger, and runs the local finger utility directing its output back over the socket to the client.

The function tcpserv in myfingerd/myfingerd.c, Section 3.6, takes a port number as an argument, binds a socket to that port, tells the kernel to listen for TCP connections on that socket, and returns the socket file descriptor number, or −1 on an error. This requires three main system calls:

- `int socket (int domain, int type, int protocol);`

  This function creates a socket, as described in Section 3.2.

- `int bind (int s, struct sockaddr *addr, int addrlen);`

  This function assigns an address to a socket, as described in Section 3.2. Unlike the finger client, which did not care about its local port number, here we specify a specific port number. INADDR_ANY can still be specified as the local IP address: on a multi-homed machine the socket will accept connections on any of the server's addresses.

  Binding a specific port number can cause complications when killing and restarting servers (for instance during debugging). Closed TCP connections can sit for a while in a state called TIME_WAIT before disappearing entirely. This can prevent a restarted TCP server from binding the same port number again, even if the old process no longer exists. The setsockopt system call shown in tcpserv avoids this problem. It tells the operating system to let the socket be bound to a port number already in use.

- `int listen (int s, int backlog);`

  listen tells the operating system to accept network connections. It returns 0 on success, and −1 on error. s is an unconnected socket bound to the port on which to accept connections. backlog specifies the number of connections the operating system accepts ahead of the application. Some modern Unix operating systems ignore that argument, but some, like FreeBSD, use it. A serious server should choose a large value, e.g., 1000. Even though there is no real resource penalty for choosing a large value, people traditionally use the value 5.

  Once you have called listen on a socket, you cannot call connect, read, or write, as the socket has no remote end. Instead, a new system call, accept, creates a new socket for each client connecting to the port s is bound to.

Once tcpserv has begun listening on a socket, main accepts connections from clients, with the system call accept.

- `int accept (int s, struct sockaddr *addr, int *addrlenp);`

  Accept takes a socket s on which one is listening and returns a new socket to which a client has just connected. If no clients have connected, accept will block until one does. accept returns −1 on an error.

  For TCP, addr should be a struct sockaddr_in *. addrlenp must be a pointer to an integer containing the value sizeof (struct sockaddr_in). accept will adjust *addrlenp to contain the actual length of the struct sockaddr it copies into

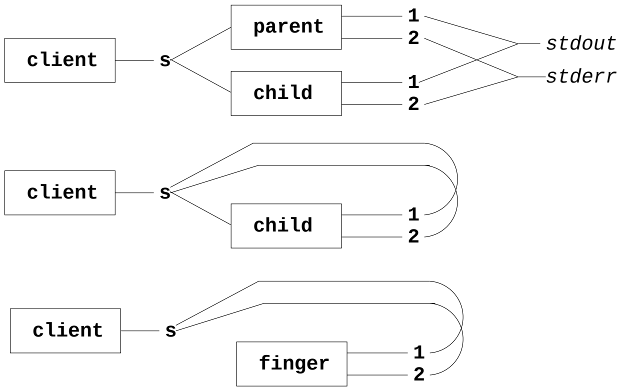


Figure 1: **file descriptors in a forking server** are inherited by the child process (top figure). In the second figure, the child process has called dup2 to connect file descriptors 1 and 2 which normally correspond to standard output and standard error to the network socket. In the final figure, the child has called execl and the local finger process runs, sending its standard output over s.

`*addr`. In the case of TCP, all `struct sockaddr_in`'s are the same size, so `*addrlenp` shouldn't change.

The finger daemon makes use of a few more Unix system calls which, while not network-specific, are often encountered in network servers. Fork it creates a new process. This new process calls dup2 to redirect its standard output and error over the accepted socket. Finally, a call to execl replaces the new process with an instance of the finger program. Finger inherits its standard output and error, so these go straight back over the network to the client.

- `int fork (void);`

  fork creates a new process, identical to the current one. fork returns twice: in the old process, fork returns a process ID of the new process. In the new or "child" process, fork returns 0. fork returns $-1$ if there is an error.

- `int dup2(int oldfd, int newfd);`

  dup2 closes file descriptor number `newfd`, and replaces it with a copy of `oldfd`. When the second argument is 1, this changes the destination of the standard output. When that argument is 2, it changes the standard error.

- `int execl(char *path, char *arg0, ..., NULL);`

  The execl system call runs a command—similar to typing it on the command line, but without shell expansion of meta-characters such as `*`, `?`, etc. The command executed will inherit all file descriptors except those with the close-on-exec flag set. execl replaces the currently executing program with the one specified by `path`. On success, it therefore doesn't return. On failure, it returns $-1$.

## 3.5 Closing a socket

If the close system call is passed the only remaining file descriptor reference to a socket, communication in both directions will be closed. If another reference to the socket exists (even in another process), communications are unaffected over the remaining descriptors. It is sometimes convenient to transmit an end-of-file over a socket without closing the socket—either because not all descriptors can be closed, or because one wishes to read from the socket even after writing an end-of-file.

Consider, for example, a protocol in which a client sends a single query and then receives a response from the server. The client might signal the end of the query with an end-of-file—effectively closing the write half of its TCP connection. Once the server receives the end-of-file, it parses and responds to the query. The client must still be able to read from the socket even after sending an end of file. It can do so using the shutdown system call.

- `int shutdown (int fd, int how);`

  shutdown shuts down communications over a socket in one or both directions, without deallocating the file descriptor and regardless of how many other file descriptor references there are to the socket. The argument `how` can be either 0, 1, or 2. 0 shuts down the socket for reading, 1 for writing, and 2 for both. When a TCP socket is shut down for writing, the process at the other end of the socket will see a 0-length read, indicating end-of-file, but data can continue to flow in the other direction.

  The TCP protocol has no way of indicating to the remote end that a socket has been shut down for reading. Thus, it is almost never useful to call shutdown on a TCP socket with a `how` argument of 0 or 2.

## 3.6 `myfingerd.c`: A simple network finger server

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <netdb.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define FINGER_PORT 79
#define FINGER_COMMAND "/usr/bin/finger"
#define bzero(ptr, size) memset (ptr, 0, size)

/* Create a TCP socket, bind it to a particular port, and call listen
 * for connections on it.  These are the three steps necessary before
 * clients can connect to a server.  */
int
```

```
tcpserv (int port)
{
  int s, n;
  struct sockaddr_in sin;

  /* The address of this server */
  bzero (&sin, sizeof (sin));
  sin.sin_family = AF_INET;
  sin.sin_port = htons (port);
  /* We are interested in listening on any and all IP addresses this
   * machine has, so use the magic IP address INADDR_ANY. */
  sin.sin_addr.s_addr = htonl (INADDR_ANY);

  s = socket (AF_INET, SOCK_STREAM, 0);
  if (s < 0) {
    perror ("socket");
    return -1;
  }

  /* Allow the program to run again even if there are old connections
   * in TIME_WAIT.  This is the magic you need to do to avoid seeing
   * "Address already in use" errors when you are killing and
   * restarting the daemon frequently.  */
  n = 1;
  if (setsockopt (s, SOL_SOCKET, SO_REUSEADDR, (char *)&n, sizeof (n)) < 0) {
    perror ("SO_REUSEADDR");
    close (s);
    return -1;
  }
  /* This function sets the close-on-exec bit of a file descriptor.
   * That way no programs we execute will inherit the TCP server file
   * descriptor. */
  fcntl (s, F_SETFD, 1);

  if (bind (s, (struct sockaddr *) &sin, sizeof (sin)) < 0) {
    fprintf (stderr, "TCP port %d: %s\n", port, strerror (errno));
    close (s);
    return -1;
  }
  if (listen (s, 5) < 0) {
    perror ("listen");
    close (s);
    return -1;
  }

  return s;
}

/* Read a line of input from a file descriptor and return it.  Returns
 * NULL on EOF/error/out of memory.  May over-read, so don't use this
 * if there is useful data after the first line. */
static char *
readline (int s)
```

```
{
  char *buf = NULL, *nbuf;
  int buf_pos = 0, buf_len = 0;
  int i, n;

  for (;;) {
    /* Ensure there is room in the buffer */
    if (buf_pos == buf_len) {
      buf_len = buf_len ? buf_len << 1 : 4;
      nbuf = realloc (buf, buf_len);
      if (!nbuf) {
        free (buf);
        return NULL;
      }
      buf = nbuf;
    }

    /* Read some data into the buffer */
    n = read (s, buf + buf_pos, buf_len - buf_pos);
    if (n <= 0) {
      if (n < 0)
        perror ("read");
      else
        fprintf (stderr, "read: EOF\n");
      free (buf);
      return NULL;
    }

    /* Look for the end of a line, and return if we got it.  Be
     * generous in what we consider to be the end of a line. */
    for (i = buf_pos; i < buf_pos + n; i++)
      if (buf[i] == '\0' || buf[i] == '\r' || buf[i] == '\n') {
        buf[i] = '\0';
        return buf;
      }

    buf_pos += n;
  }
}

static void
runfinger (int s)
{
  char *user;

  /* Read the username being fingered. */
  user = readline (s);

  /* Now connect standard output and standard error to the socket,
   * instead of the invoking user's terminal. */
  if (dup2 (s, 1) < 0 || dup2 (s, 2) < 0) {
    perror ("dup2");
    exit (1);
```

```
  }
  close (s);

  /* Run the finger command.  It will inherit our standard output and
   * error, and therefore send its results back over the network. */
  execl (FINGER_COMMAND, "finger", "--", *user ? user : NULL, NULL);

  /* We should never get here, unless we couldn't run finger. */
  perror (FINGER_COMMAND);
  exit (1);
}

int
main (int argc, char **argv)
{
  int ss, cs;
  struct sockaddr_in sin;
  int sinlen;
  int pid;

  /* This system call allows one to call fork without worrying about
   * calling wait.  Don't worry about what it means unless you start
   * caring about the exit status of forked processes, in which case
   * you should delete this line and read the manual pages for wait
   * and waitpid.  For a description of what this signal call really
   * does, see the manual page for sigaction and look for
   * SA_NOCLDWAIT.  Signal is an older signal interface which when
   * invoked this way is equivalent to setting SA_NOCLDWAIT. */
  signal (SIGCHLD, SIG_IGN);

  ss = tcpserv (FINGER_PORT);
  if (ss < 0)
    exit (1);

  for (;;) {
    sinlen = sizeof (sin);
    cs = accept (ss, (struct sockaddr *) &sin, &sinlen);
    if (cs < 0) {
      perror ("accept");
      exit (1);
    }
    printf ("connection from %s\n", inet_ntoa (sin.sin_addr));

    pid = fork ();
    if (!pid)
      /* Child process */
      runfinger (cs);

    close (cs);
  }
}
```