# 60

# SOCKETS: SERVER DESIGN

This chapter discusses the fundamentals of designing iterative and concurrent servers and describes *inetd*, a special daemon designed to facilitate the creation of Internet servers.

## 60.1 Iterative and Concurrent Servers

Two common designs for network servers using sockets are the following:

- *Iterative*: The server handles one client at a time, processing that client's request(s) completely, before proceeding to the next client.
- *Concurrent*: The server is designed to handle multiple clients simultaneously.

We have already seen an example of an iterative server using FIFOs in Section 44.8 and an example of a concurrent server using System V message queues in Section 46.8.

Iterative servers are usually suitable only when client requests can be handled quickly, since each client must wait until all of the preceding clients have been serviced. A typical scenario for employing an iterative server is where the client and server exchange a single request and response.

Concurrent servers are suitable when a significant amount of processing time is required to handle each request, or where the client and server engage in an extended conversation, passing messages back and forth. In this chapter, we mainly focus on the traditional (and simplest) method of designing a concurrent server: creating a new child process for each new client. Each server child performs all tasks necessary to service a single client and then terminates. Since each of these processes can operate independently, multiple clients can be handled simultaneously. The principal task of the main server process (the parent) is to create a new child process for each new client. (A variation on this approach is to create a new thread for each client.)

In the following sections, we look at examples of an iterative and a concurrent server using Internet domain sockets. These two servers implement the *echo* service (RFC 862), a rudimentary service that returns a copy of whatever the client sends it.

## 60.2 An Iterative UDP *echo* Server

In this and the next section, we present servers for the *echo* service. The *echo* service operates on both UDP and TCP port 7. (Since this is a reserved port, the *echo* server must be run with superuser privileges.)

The UDP *echo* server continuously reads datagrams, returning a copy of each datagram to the sender. Since the server needs to handle only a single message at a time, an iterative server design suffices. The header file for the server is shown in Listing 60-1.

**Listing 60-1:** Header file for `id_echo_sv.c` and `id_echo_cl.c`

```
                                                          sockets/id_echo.h
#include "inet_sockets.h"       /* Declares our socket functions */
#include "tlpi_hdr.h"

#define SERVICE "echo"          /* Name of UDP service */

#define BUF_SIZE 500            /* Maximum size of datagrams that can
                                   be read by client and server */
                                                          sockets/id_echo.h
```

Listing 60-2 shows the implementation of the server. Note the following points regarding the server implementation:

- We use the *becomeDaemon()* function of Section 37.2 to turn the server into a daemon.
- To shorten this program, we employ the Internet domain sockets library developed in Section 59.12.
- If the server can't send a reply to the client, it logs a message using *syslog()*.

> In a real-world application, we would probably apply some rate limit to the messages written with *syslog()*, both to prevent the possibility of an attacker filling the system log and because each call to *syslog()* is expensive, since (by default) *syslog()* in turn calls *fsync()*.

**Listing 60-2:** An iterative server that implements the UDP *echo* service

sockets/id_echo_sv.c

```
#include <syslog.h>
#include "id_echo.h"
#include "become_daemon.h"

int
main(int argc, char *argv[])
{
    int sfd;
    ssize_t numRead;
    socklen_t addrlen, len;
    struct sockaddr_storage claddr;
    char buf[BUF_SIZE];
    char addrStr[IS_ADDR_STR_LEN];

    if (becomeDaemon(0) == -1)
        errExit("becomeDaemon");

    sfd = inetBind(SERVICE, SOCK_DGRAM, &addrlen);
    if (sfd == -1) {
        syslog(LOG_ERR, "Could not create server socket (%s)", strerror(errno));
        exit(EXIT_FAILURE);
    }

    /* Receive datagrams and return copies to senders */

    for (;;) {
        len = sizeof(struct sockaddr_storage);
        numRead = recvfrom(sfd, buf, BUF_SIZE, 0,
                           (struct sockaddr *) &claddr, &len);
        if (numRead == -1)
            errExit("recvfrom");

        if (sendto(sfd, buf, numRead, 0, (struct sockaddr *) &claddr, len)
                != numRead)
            syslog(LOG_WARNING, "Error echoing response to %s (%s)",
                    inetAddressStr((struct sockaddr *) &claddr, len,
                                   addrStr, IS_ADDR_STR_LEN),
                    strerror(errno));
    }
}
```

sockets/id_echo_sv.c

To test the server, we use the client program shown in Listing 60-3. This program also employs the Internet domain sockets library developed in Section 59.12. As its first command-line argument, the client program expects the name of the host on which the server resides. The client executes a loop in which it sends each of its remaining command-line arguments to the server as separate datagrams, and reads and prints each response datagram sent back by the server.

**Listing 60-3:** A client for the UDP *echo* service

```
#include "id_echo.h"

int
main(int argc, char *argv[])
{
    int sfd, j;
    size_t len;
    ssize_t numRead;
    char buf[BUF_SIZE];

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s: host msg...\n", argv[0]);

    /* Construct server address from first command-line argument */

    sfd = inetConnect(argv[1], SERVICE, SOCK_DGRAM);
    if (sfd == -1)
        fatal("Could not connect to server socket");

    /* Send remaining command-line arguments to server as separate datagrams */

    for (j = 2; j < argc; j++) {
        len = strlen(argv[j]);
        if (write(sfd, argv[j], len) != len)
            fatal("partial/failed write");

        numRead = read(sfd, buf, BUF_SIZE);
        if (numRead == -1)
            errExit("read");

        printf("[%ld bytes] %.*s\n", (long) numRead, (int) numRead, buf);
    }

    exit(EXIT_SUCCESS);
}
```

**sockets/id_echo_cl.c**

Here is an example of what we see when we run the server and two instances of the client:

```
$ su                                    Need privilege to bind reserved port
Password:
# ./id_echo_sv                          Server places itself in background
# exit                                  Cease to be superuser
$ ./id_echo_cl localhost hello world    This client sends two datagrams
[5 bytes] hello                         Client prints responses from server
[5 bytes] world
$ ./id_echo_cl localhost goodbye        This client sends one datagram
[7 bytes] goodbye
```

## 60.3 A Concurrent TCP *echo* Server

The TCP *echo* service also operates on port 7. The TCP *echo* server accepts a connection and then loops continuously, reading all transmitted data and sending it back to the client on the same socket. The server continues reading until it detects end-of-file, at which point it closes its socket (so that the client sees end-of-file if it is still reading from its socket).

Since the client may send an indefinite amount of data to the server (and thus servicing the client may take an indefinite amount of time), a concurrent server design is appropriate, so that multiple clients can be simultaneously served. The server implementation is shown in Listing 60-4. (We show an implementation of a client for this service in Section 61.2.) Note the following points about the implementation:

- The server becomes a daemon by calling the *becomeDaemon()* function shown in Section 37.2.
- To shorten this program, we employ the Internet domain sockets library shown in Listing 59-9 (page 1228).
- Since the server creates a child process for each client connection, we must ensure that zombies are reaped. We do this within a `SIGCHLD` handler.
- The main body of the server consists of a `for` loop that accepts a client connection and then uses *fork()* to create a child process that invokes the *handleRequest()* function to handle that client. In the meantime, the parent continues around the `for` loop to accept the next client connection.

  > In a real-world application, we would probably include some code in our server to place an upper limit on the number of child processes that the server could create, in order to prevent an attacker from attempting a remote fork bomb by using the service to create so many processes on the system that it becomes unusable. We could impose this limit by adding extra code in the server to count the number of children currently executing (this count would be incremented after a successful *fork()* and decremented as each child was reaped in the `SIGCHLD` handler). If the limit on the number of children were reached, we could then temporarily stop accepting connections (or alternatively, accept connections and then immediately close them).

- After each *fork()*, the file descriptors for the listening and connected sockets are duplicated in the child (Section 24.2.1). This means that both the parent and the child could communicate with the client using the connected socket. However, only the child needs to perform such communication, and so the parent closes the file descriptor for the connected socket immediately after the *fork()*. (If the parent did not do this, then the socket would never actually be closed; furthermore, the parent would eventually run out of file descriptors.) Since the child doesn't accept new connections, it closes its duplicate of the file descriptor for the listening socket.
- Each child process terminates after handling a single client.

**Listing 60-4:** A concurrent server that implements the TCP *echo* service

**sockets/is_echo_sv.c**

```
#include <signal.h>
#include <syslog.h>
#include <sys/wait.h>
#include "become_daemon.h"
#include "inet_sockets.h"       /* Declarations of inet*() socket functions */
#include "tlpi_hdr.h"

#define SERVICE "echo"          /* Name of TCP service */
#define BUF_SIZE 4096

static void             /* SIGCHLD handler to reap dead child processes */
grimReaper(int sig)
{
    int savedErrno;             /* Save 'errno' in case changed here */

    savedErrno = errno;
    while (waitpid(-1, NULL, WNOHANG) > 0)
        continue;
    errno = savedErrno;
}

/* Handle a client request: copy socket input back to socket */

static void
handleRequest(int cfd)
{
    char buf[BUF_SIZE];
    ssize_t numRead;

    while ((numRead = read(cfd, buf, BUF_SIZE)) > 0) {
        if (write(cfd, buf, numRead) != numRead) {
            syslog(LOG_ERR, "write() failed: %s", strerror(errno));
            exit(EXIT_FAILURE);
        }
    }

    if (numRead == -1) {
        syslog(LOG_ERR, "Error from read(): %s", strerror(errno));
        exit(EXIT_FAILURE);
    }
}

int
main(int argc, char *argv[])
{
    int lfd, cfd;               /* Listening and connected sockets */
    struct sigaction sa;

    if (becomeDaemon(0) == -1)
        errExit("becomeDaemon");
```

```
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART;
    sa.sa_handler = grimReaper;
    if (sigaction(SIGCHLD, &sa, NULL) == -1) {
        syslog(LOG_ERR, "Error from sigaction(): %s", strerror(errno));
        exit(EXIT_FAILURE);
    }

    lfd = inetListen(SERVICE, 10, NULL);
    if (lfd == -1) {
        syslog(LOG_ERR, "Could not create server socket (%s)", strerror(errno));
        exit(EXIT_FAILURE);
    }

    for (;;) {
        cfd = accept(lfd, NULL, NULL);  /* Wait for connection */
        if (cfd == -1) {
            syslog(LOG_ERR, "Failure in accept(): %s", strerror(errno));
            exit(EXIT_FAILURE);
        }

        /* Handle each client request in a new child process */

        switch (fork()) {
        case -1:
            syslog(LOG_ERR, "Can't create child (%s)", strerror(errno));
            close(cfd);                 /* Give up on this client */
            break;                      /* May be temporary; try next client */

        case 0:                         /* Child */
            close(lfd);                 /* Unneeded copy of listening socket */
            handleRequest(cfd);
            _exit(EXIT_SUCCESS);

        default:                        /* Parent */
            close(cfd);                 /* Unneeded copy of connected socket */
            break;                      /* Loop to accept next connection */
        }
    }
}
```

**sockets/is_echo_sv.c**

## 60.4 Other Concurrent Server Designs

The traditional concurrent server model described in the previous section is adequate for many applications that need to simultaneously handle multiple clients via TCP connections. However, for very high-load servers (for example, web servers handling thousands of requests per minute), the cost of creating a new child (or even thread) for each client imposes a significant burden on the server (refer to Section 28.3), and alternative designs need to be employed. We briefly consider some of these alternatives.

### Preforked and prethreaded servers

Preforked and prethreaded servers are described in some detail in Chapter 30 of [Stevens et al., 2004]. The key ideas are the following:

- Instead of creating a new child process (or thread) for each client, the server precreates a fixed number of child processes (or threads) immediately on startup (i.e., before any client requests are even received). These children constitute a so-called *server pool*.
- Each child in the server pool handles one client at a time, but instead of terminating after handling the client, the child fetches the next client to be serviced and services it, and so on.

Employing the above technique requires some careful management within the server application. The server pool should be large enough to ensure adequate response to client requests. This means that the server parent must monitor the number of unoccupied children, and, in times of peak load, increase the size of the pool so that there are always enough child processes available to immediately serve new clients. If the load decreases, then the size of the server pool should be reduced, since having excess processes on the system can degrade overall system performance.

In addition, the children in the server pool must follow some protocol to allow them to exclusively select individual client connections. On most UNIX implementations (including Linux), it is sufficient to have each child in the pool block in an *accept()* call on the listening descriptor. In other words, the server parent creates the listening socket before creating any children, and each of the children inherits a file descriptor for the socket during the *fork()*. When a new client connection arrives, only one of the children will complete the *accept()* call. However, because *accept()* is not an atomic system call on some older implementations, the call may need to be bracketed by some mutual-exclusion technique (e.g., a file lock) to ensure that only one child at a time performs the call ([Stevens et al., 2004]).

> There are alternatives to having all of the children in the server pool perform *accept()* calls. If the server pool consists of separate processes, the server parent can perform the *accept()* call, and then pass the file descriptor containing the new connection to one of the free processes in the pool, using a technique that we briefly describe in Section 61.13.3. If the server pool consists of threads, the main thread can perform the *accept()* call, and then inform one of the free server threads that a new client is available on the connected descriptor.

### Handling multiple clients from a single process

In some cases, we can design a single server process to handle multiple clients. To do this, we must employ one of the I/O models (I/O multiplexing, signal-driven I/O, or *epoll*) that allow a single process to simultaneously monitor multiple file descriptors for I/O events. These models are described in Chapter 63.

In a single-server design, the server process must take on some of the scheduling tasks that are normally handled by the kernel. In a solution that involves one server process per client, we can rely on the kernel to ensure that each server process (and thus client) gets a fair share of access to the resources of the server host. But when we use a single server process to handle multiple clients, the server must do some work

to ensure that one or a few clients don't monopolize access to the server while other clients are starved. We say a little more about this point in Section 63.4.6.

### Using server farms

Other approaches to handling high client loads involve the use of multiple server systems—a *server farm*.

One of the simplest approaches to building a server farm (employed by some web servers) is *DNS round-robin load sharing* (or *load distribution*), where the authoritative name server for a zone maps the same domain name to several IP addresses (i.e., several servers share the same domain name). Successive requests to the DNS server to resolve the domain name return these IP addresses in a different order, in a round-robin fashion. Further information about DNS round-robin load sharing can be found in [Albitz & Liu, 2006].

Round-robin DNS has the advantage of being inexpensive and easy to set up. However, it does present some problems. One of these is the caching performed by remote DNS servers, which means that future requests from clients on a particular host (or set of hosts) bypass the round-robin DNS server and are always handled by the same server. Also, round-robin DNS doesn't have any built-in mechanisms for ensuring good load balancing (different clients may place different loads on a server) or ensuring high availability (what if one of the servers dies or the server application that it is running crashes?). Another issue that we may need to consider—one that is faced by many designs that employ multiple server machines—is ensuring *server affinity*; that is, ensuring that a sequence of requests from the same client are all directed to the same server, so that any state information maintained by the server about the client remains accurate.

A more flexible, but also more complex, solution is *server load balancing*. In this scenario, a single load-balancing server routes incoming client requests to one of the members of the server farm. (To ensure high availability, there may be a backup server that takes over if the primary load-balancing server crashes.) This eliminates the problems associated with remote DNS caching, since the server farm presents a single IP address (that of the load-balancing server) to the outside world. The load-balancing server incorporates algorithms to measure or estimate server load (perhaps based on metrics supplied by the members of the server farm) and intelligently distribute the load across the members of the server farm. The load-balancing server also automatically detects failures in members of the server farm (and the addition of new servers, if demand requires it). Finally, a load-balancing server may also provide support for server affinity. Further information about server load balancing can be found in [Kopparapu, 2002].

## 60.5 The *inetd* (Internet Superserver) Daemon

If we look through the contents of `/etc/services`, we see literally hundreds of different services listed. This implies that a system could theoretically be running a large number server processes. However, most of these servers would usually be doing nothing but waiting for infrequent connection requests or datagrams. All of these server processes would nevertheless occupy slots in the kernel process table, and consume some memory and swap space, thus placing a load on the system.

The *inetd* daemon is designed to eliminate the need to run large numbers of infrequently used servers. Using *inetd* provides two main benefits:

- Instead of running a separate daemon for each service, a single process—the *inetd* daemon—monitors a specified set of socket ports and starts other servers as required. Thus, the number of processes running on the system is reduced.
- The programming of the servers started by *inetd* is simplified, because *inetd* performs several of the steps that are commonly required by all network servers on startup.

Since it oversees a range of services, invoking other servers as required, *inetd* is sometimes known as the *Internet superserver*.

> An extended version of *inetd*, *xinetd*, is provided in some Linux distributions. Among other things, *xinetd* adds a number of security enhancements. Information about *xinetd* can be found at *http://www.xinetd.org/*.

### Operation of the *inetd* daemon

The *inetd* daemon is normally started during system boot. After becoming a daemon process (Section 37.2), *inetd* performs the following steps:

1. For each of the services specified in its configuration file, `/etc/inetd.conf`, *inetd* creates a socket of the appropriate type (i.e., stream or datagram) and binds it to the specified port. Each TCP socket is additionally marked to permit incoming connections via a call to *listen()*.
2. Using the *select()* system call (Section 63.2.1), *inetd* monitors all of the sockets created in the preceding step for datagrams or incoming connection requests.
3. The *select()* call blocks until either a UDP socket has a datagram available to read or a connection request is received on a TCP socket. In the case of a TCP connection, *inetd* performs an *accept()* for the connection before proceeding to the next step.
4. To start the server specified for this socket, *inetd()* calls *fork()* to create a new process that then does an *exec()* to start the server program. Before performing the *exec()*, the child process performs the following steps:
   a) Close all of the file descriptors inherited from its parent, except the one for the socket on which the UDP datagram is available or the TCP connection has been accepted.
   b) Use the techniques described in Section 5.5 to duplicate the socket file descriptor on file descriptors 0, 1, and 2, and close the socket file descriptor itself (since it is no longer required). After this step, the execed server is able to communicate on the socket by using the three standard file descriptors.
   c) Optionally, set the user and group IDs for the execed server to values specified in `/etc/inetd.conf`.
5. If a connection was accepted on a TCP socket in step 3, *inetd* closes the connected socket (since it is needed only in the execed server).
6. The *inetd* server returns to step 2.

### The /etc/inetd.conf file

The operation of the *inetd* daemon is controlled by a configuration file, normally /etc/inetd.conf. Each line in this file describes one of the services to be handled by *inetd*. Listing 60-5 shows some examples of entries in the /etc/inetd.conf file that comes with one Linux distribution.

**Listing 60-5:** Example lines from /etc/inetd.conf

```
# echo  stream  tcp  nowait  root    internal
# echo  dgram   udp  wait    root    internal
ftp     stream  tcp  nowait  root    /usr/sbin/tcpd   in.ftpd
telnet  stream  tcp  nowait  root    /usr/sbin/tcpd   in.telnetd
login   stream  tcp  nowait  root    /usr/sbin/tcpd   in.rlogind
```

The first two lines of Listing 60-5 are commented out by the initial # character; we show them now since we'll refer to the *echo* service shortly.

Each line of /etc/inetd.conf consists of the following fields, delimited by white space:

- *Service name*: This specifies the name of a service from the /etc/services file. In conjunction with the *protocol* field, this is used to look up /etc/services to determine which port number *inetd* should monitor for this service.
- *Socket type*: This specifies the type of socket used by this service—for example, stream or dgram.
- *Protocol*: This specifies the protocol to be used by this socket. This field can contain any of the Internet protocols listed in the file /etc/protocols (documented in the *protocols(5)* manual page), but almost every service specifies either tcp (for TCP) or udp (for UDP).
- *Flags*: This field contains either wait or nowait. This field specifies whether or not the server execed by *inetd* (temporarily) takes over management of the socket for this service. If the execed server manages the socket, then this field is specified as wait. This causes *inetd* to remove this socket from the file descriptor set that it monitors using *select()* until the execed server exits (*inetd* detects this via a handler for SIGCHLD). We say some more about this field below.
- *Login name*: This field consists of a username from /etc/passwd, optionally followed by a period (.) and a group name from /etc/group. These determine the user and group IDs under which the execed server is run. (Since *inetd* runs with an effective user ID of *root*, its children are also privileged and can thus use calls to *setuid()* and *setgid()* to change process credentials if desired.)
- *Server program*: This specifies the pathname of the server program to be execed.
- *Server program arguments*: This field specifies one or more arguments, separated by white space, to be used as the argument list when execing the server program. The first of these corresponds to *argv[0]* in the execed program and is thus usually the same as the basename part of the *server program* name. The next argument corresponds to *argv[1]*, and so on.

> In the example lines shown in Listing 60-5 for the *ftp*, *telnet*, and *login* services, we see the server program and arguments are set up differently than just described. All three of these services cause *inetd* to invoke the same program, *tcpd(8)* (the TCP daemon wrapper), which performs some logging and access-control checks before in turn execing the appropriate program, based on the value specified as the first server program argument (which is available to *tcpd* via *argv[0]*). Further information about *tcpd* can be found in the *tcpd(8)* manual page and in [Mann & Mitchell, 2003].

Stream socket (TCP) servers invoked by *inetd* are normally designed to handle just a single client connection and then terminate, leaving *inetd* with the job of listening for further connections. For such servers, *flags* should be specified as `nowait`. (If, instead, the execed server is to accept connections, then `wait` should be specified, in which case *inetd* does not accept the connection, but instead passes the file descriptor for the *listening* socket to the execed server as descriptor 0.)

For most UDP servers, the *flags* field should be specified as `wait`. A UDP server invoked by *inetd* is normally designed to read and process all outstanding datagrams on the socket and then terminate. (This usually requires some sort of timeout when reading the socket, so that the server terminates when no new datagrams arrive within a specified interval.) By specifying `wait`, we prevent the *inetd* daemon from simultaneously trying to *select()* on the socket, which would have the unintended consequence that *inetd* would race the UDP server to check for datagrams and, if it won the race, start another instance of the UDP server.

> Because the operation of *inetd* and the format of its configuration file are not specified by SUSv3, there are some (generally small) variations in the values that can be specified in the fields of `/etc/inetd.conf`. Most versions of *inetd* provide at least the syntax that we describe in the main text. For further details, see the *inetd.conf(8)* manual page.

As an efficiency measure, *inetd* implements a few simple services itself, instead of execing separate servers to perform the task. The UDP and TCP *echo* services are examples of services that *inetd* implements. For such services, the *server program* field of the corresponding `/etc/inetd.conf` record is specified as `internal`, and the *server program arguments* are omitted. (In the example lines in Listing 60-5, we saw that the *echo* service entries were commented out. To enable the *echo* service, we need to remove the # character at the start of these lines.)

Whenever we change the `/etc/inetd.conf` file, we need to send a `SIGHUP` signal to *inetd* to request it to reread the file:

```
# killall -HUP inetd
```

### Example: invoking a TCP *echo* service via *inetd*

We noted earlier that *inetd* simplifies the programming of servers, especially concurrent (usually TCP) servers. It does this by carrying out the following steps on behalf of the servers it invokes:

1. Perform all socket-related initialization, calling *socket()*, *bind()*, and (for TCP servers) *listen()*.
2. For a TCP service, perform an *accept()* for the new connection.

3. Create a new process to handle the incoming UDP datagram or TCP connection. The process is automatically set up as a daemon. The *inetd* program handles all details of process creation via *fork()* and the reaping of dead children via a handler for SIGCHLD.
4. Duplicate the file descriptor of the UDP socket or the connected TCP socket on file descriptors 0, 1, and 2, and close all other file descriptors (since they are unused in the execed server).
5. Exec the server program.

(In the description of the above steps, we assume the usual cases that the *flags* field of the service entry in /etc/inetd.conf is specified as nowait for TCP services and wait for UDP services.)

As an example of how *inetd* simplifies the programming of a TCP service, in Listing 60-6, we show the *inetd*-invoked equivalent of the TCP *echo* server from Listing 60-4. Since *inetd* performs all of the above steps, all that remains of the server is the code executed by the child process to handle the client request, which can be read from file descriptor 0 (STDIN_FILENO).

If the server resides in the directory /bin (for example), then we would need to create the following entry in /etc/inetd.conf in order to have *inetd* invoke the server:

```
echo stream tcp nowait root /bin/is_echo_inetd_sv is_echo_inetd_sv
```

**Listing 60-6:** TCP *echo* server designed to be invoked via *inetd*

**sockets/is_echo_inetd_sv.c**

```
#include <syslog.h>
#include "tlpi_hdr.h"

#define BUF_SIZE 4096

int
main(int argc, char *argv[])
{
    char buf[BUF_SIZE];
    ssize_t numRead;

    while ((numRead = read(STDIN_FILENO, buf, BUF_SIZE)) > 0) {
        if (write(STDOUT_FILENO, buf, numRead) != numRead) {
            syslog(LOG_ERR, "write() failed: %s", strerror(errno));
            exit(EXIT_FAILURE);
        }
    }

    if (numRead == -1) {
        syslog(LOG_ERR, "Error from read(): %s", strerror(errno));
        exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

**sockets/is_echo_inetd_sv.c**

## 60.6 Summary

An iterative server handles one client at a time, processing that client's request(s) completely, before proceeding to the next client. A concurrent server handles multiple clients simultaneously. In high-load scenarios, a traditional concurrent server design that creates a new child process (or thread) for each client may not perform well enough, and we outlined a range of other approaches for concurrently handling large numbers of clients.

The Internet superserver daemon, *inetd*, monitors multiple sockets and starts the appropriate servers in response to incoming UDP datagrams or TCP connections. Using *inetd* allows us to decrease system load by minimizing the number of network server processes on the system, and also simplifies the programming of server processes, since it performs most of the initialization steps required by a server.

#### Further information

Refer to the sources of further information listed in Section 59.15.

## 60.7 Exercises

**60-1.** Add code to the program in Listing 60-4 (`is_echo_sv.c`) to place a limit on the number of simultaneously executing children.

**60-2.** Sometimes, it may be necessary to write a socket server so that it can be invoked either directly from the command line or indirectly via *inetd*. In this case, a command-line option is used to distinguish the two cases. Modify the program in Listing 60-4 so that, if it is given a *−i* command-line option, it assumes that it is being invoked by *inetd* and handles a single client on the connected socket, which *inetd* supplies via `STDIN_FILENO`. If the *−i* option is not supplied, then the program can assume it is being invoked from the command line, and operate in the usual fashion. (This change requires only the addition of a few lines of code.) Modify `/etc/inetd.conf` to invoke this program for the *echo* service.