

chapter 6

Beyond Basic Socket Programming

Our client and server examples demonstrate the basic model for socket programming. The next step is to integrate these ideas into various programming models such as multi-tasking, signalling, and broadcasting. We demonstrate these principles in the context of standard UNIX programming; however, most modern operating systems support similar features (e.g., processes and threads).

6.1 Socket Options

The TCP/IP protocol developers spent a good deal of time thinking about the default behaviors that would satisfy most applications. (If you doubt this, read RFCs 1122 and 1123, which describe in excruciating detail the recommended behaviors—based on years of experience—for implementations of the TCP/IP protocols.) For most applications, the designers did a good job; however, it is seldom the case that “one-size-fits-all” really fits all. For example, each socket has an associated receive buffer. How big should it be? Each implementation has a default size; however, this value may not always be appropriate for your application (see also Section 7.1). This particular aspect of a socket’s behavior, along with many others, is associated with a *socket option*: You can change the receive buffer size of a socket by modifying the value of the associated socket option. The functions `getsockopt()` and `setsockopt()` allow socket option values to be queried and set, respectively.

```
int getsockopt(int socket, int level, int optName, void *optVal, socklen_t *optLen)
int setsockopt(int socket, int level, int optName, const void *optVal, socklen_t optLen)
```

For both functions, *socket* must be a socket descriptor allocated by *socket()*. The available socket options are divided into levels that correspond to the layers of the protocol stack; the second parameter indicates the level of the option in question. Some options are protocol independent and are thus handled by the socket layer itself (*SOL_SOCKET*), some are specific to the transport protocol (*IPPROTO_TCP*), and some are handled by the internetwork protocol (*IPPROTO_IP*). The option itself is specified by the integer *optName*, which is always specified using a system-defined constant. The parameter *optVal* is a pointer to a buffer. For *getsockopt()*, the option's current value will be placed in that buffer by the implementation, whereas for *setsockopt()*, the socket option in the implementation will be set to the value in the buffer. In both calls, *optLen* specifies the length of the buffer, which must be correct for the particular option in question. Note that in *getsockopt()*, *optLen* is an in-out parameter, initially pointing to an integer containing the size of the buffer; on return the pointed-to integer contains the size of the option value. The following code segment demonstrates how to fetch and then double the configured size (in bytes) of the socket's receive buffer:

```
int rcvBufferSize;
// Retrieve and print the default buffer size
int sockOptSize = sizeof(rcvBufferSize);
if (getsockopt(sock, SOL_SOCKET, SO_RCVBUF, &rcvBufferSize, &sockOptSize) < 0)
    DieWithSystemMessage("getsockopt() failed");
printf("Initial Receive Buffer Size: %d\n", rcvBufferSize);

// Double the buffer size
rcvBufferSize *= 2;
if (setsockopt(sock, SOL_SOCKET, SO_RCVBUF, &rcvBufferSize, sizeof(rcvBufferSize)) < 0)
    DieWithSystemMessage("setsockopt() failed");
```

Note that value passed to *setsockopt()* is *not* guaranteed to be the new size of the socket buffer, even if the call apparently succeeds. Rather, it is best thought of as a “hint” to the system about the value desired by the user; the system, after all, has to manage resources for *all* users and may consider other factors in adjusting buffer size.

Table 6.1 shows some commonly used options at each level, including a description and the data type of the buffer pointed to by *optVal*.

6.2 Signals

Signals provide a mechanism for notifying programs that certain events have occurred—for example, the user typed the “interrupt” character, or a timer expired. Some of the events (and therefore the notification) may occur *asynchronously*, which means that the notification

SOL_SOCKET optname	Type	Values	Description
SO_BROADCAST	int	0,1	Broadcast allowed
SO_KEEPALIVE	int	0,1	Keepalive messages enabled (if implemented by protocol)
SO_LINGER	linger{}	time	Time to delay close return waiting for confirmation (see Section 7.4)
SO_RCVBUF	int	bytes	Bytes in the socket receive buffer (see code on page 100 and Section 7.1)
SO_RCVLOWAT	int	bytes	Minimum number of available bytes that will cause recv to return
SO_REUSEADDR	int	0,1	Binding allowed (under certain conditions) to an address or port already in use (see Section 7.4)
SO_SNDLOWAT	int	bytes	Minimum bytes to send
SO_SNDBUF	int	bytes	Bytes in the socket send buffer (see Section 7.1)
IPPROTO_TCP optname	type	Values	Description
TCP_MAX	int	seconds	Seconds between keepalive messages.
TCP_NODELAY	int	0, 1	Disallow delay from Nagle's algorithm for data merging
IPPROTO_IP optname	type	Values	Description
IP_TTL	int	0 - 255	Time-to-live for unicast IP packets.
IP_MULTICAST_TTL	unsigned char	0 - 255	Time-to-live for multicast IP packets (see MulticastSender.c on page 138)
IP_MULTICAST_LOOP	int	0,1	Enables multicast socket to receive packets it sent
IP_ADD_MEMBERSHIP	ip_mreq{}	group address	Enables reception of packets addressed to the specified multicast group (see MulticastReceiver.c on page 141) – Set only
IP_DROP_MEMBERSHIP	ip_mreq{}	group address	Disables reception of packets addressed to the specified multicast group – Set only

Table 6.1: *continued*

IPPROTO_IPV6 optname	type	Values	Description
IPV6_V6ONLY	int	0,1	Restrict IPv6 sockets to only IPv6 communication.
IPV6_UNICAST_HOPS	int	-1 - 255	Time-to-live for unicast IP packets.
IPV6_MULTICAST_HOPS	int	-1 - 255	Time-to-live for multicast IP packets (see MulticastSender.c on page 138)
IPV6_MULTICAST_LOOP	u_int	0,1	Enables multicast socket to receive packets it sent
IPV6_JOIN_GROUP	ipv6_mreq{}	group address	Enables reception of packets addressed to the specified multicast group (see MulticastReceiver.c on page 141) – Set only
IPV6_LEAVE_GROUP	ipv6_mreq{}	group address	Disables reception of packets addressed to the specified multicast group – Set only

Table 6.1: Socket Options

is delivered to the program regardless of which statement it is executing. When a signal is delivered to a running program, one of four things happens:

1. The signal is ignored. The process is never aware that the signal was delivered.
2. The program is forcibly terminated by the operating system.
3. Program execution is interrupted and a *signal-handling routine*, specified by (and part of) the program, is executed. This execution happens in a different thread of control from the main thread(s) of the program so that the program is not necessarily immediately aware of it.
4. The signal is *blocked*, that is, prevented from having any effect until the program takes action to allow its delivery. Each process has a *mask*, indicating which signals are currently blocked in that process. (Actually, each thread in a program can have its own signal mask.)

UNIX has dozens of different signals, each indicating the occurrence of a different type of event. Each signal has a system-defined *default behavior*, which is one of the first two possibilities listed above. For example, termination is the default behavior for SIGINT, which is delivered when the interrupt character (usually Control-C) is received via the controlling terminal for that process.

Signals are a complicated animal, and a full treatment is beyond the scope of this book. However, some signals are frequently encountered in the context of socket programming.

Moreover, **any program that sends on a TCP socket must explicitly deal with SIGPIPE in order to be robust.** Therefore, we present the basics of dealing with signals, focusing on these five:

Type	Triggering Event	Default
SIGALRM	Expiration of an alarm timer	termination
SIGCHLD	Child process exit	ignore
SIGINT	Interrupt char (Control-C) input	termination
SIGIO	Socket ready for I/O	ignore
SIGPIPE	Attempt to write to a closed socket	termination

An application program can change the default behavior¹ for a particular signal using `sigaction()`:

```
int sigaction(int whichSignal, const struct sigaction *newAction, struct sigaction *oldAction)
```

`sigaction()` returns 0 on success and `-1` on failure; details of its semantics, however, are a bit more involved.

Each signal is identified by an integer constant; *whichSignal* specifies the signal for which the behavior is being changed. The *newAction* parameter points to a **sigaction** structure that defines the new behavior for the given signal type; if the pointer *oldAction* is non-null, a **sigaction** structure describing the previous behavior for the given signal is copied into it, as shown here:

```
struct sigaction {
    void (*sa_handler)(int); // Signal handler
    sigset_t sa_mask;        // Signals to be blocked during handler execution
    int sa_flags;             // Flags to modify default behavior
};
```

The field `sa_handler` (of type “pointer to function of one integer parameter that returns void”) controls which of the first three possibilities occurs when a signal is delivered (i.e., when it is not masked). If its value is the special constant `SIG_IGN`, the signal will be ignored. If its value is `SIG_DFL`, the default behavior for that signal will be used. If its value is the address of a function (which is guaranteed to be different from the two constants), that function will be invoked with a parameter indicating the signal that was delivered. (If the same handler function

¹For some signals, the default behavior cannot be changed, nor can the signal be blocked; however, this is not true for any of the five we consider.

is used for multiple signals, the parameter can be used to determine which one caused the invocation.)

Signals can be “nested” in the following sense: While one signal is being handled, another is delivered. As you can imagine, this can get rather complicated. Fortunately, the `sigaction()` mechanism allows some signals to be temporarily blocked (in addition to those that are already blocked by the process’s signal mask) while the specified signal is handled. The field `sa_mask` specifies the signals to be blocked while handling *whichSignal*; it is only meaningful when `sa_handler` is not `SIG_IGN` or `SIG_DFL`. By default *whichSignal* is always blocked regardless of whether it is reflected in `sa_mask`. (On some systems, setting the flag `SA_NODEFER` in `sa_flags` allows the specified signal to be delivered while it is being handled.) The `sa_flags` field controls some further details of the way *whichSignal* is handled; these details are beyond the scope of this discussion.

`sa_mask` is implemented as a set of boolean flags, one for each type of signal. This set of flags can be manipulated with the following four functions:

```
int sigemptyset(sigset_t *set)
int sigfillset(sigset_t *set)
int sigaddset(sigset_t *set, int whichSignal)
int sigdelset(sigset_t *set, int whichSignal)
```

`sigfillset()` and `sigemptyset()` set and unset all of the flags in the given set. `sigaddset()` and `sigdelset()` set and unset individual flags, specified by the signal number, in the given set. All four functions return 0 for success and `-1` for failure.

`SigAction.c` shows a simple `sigaction()` example to provide a handler for `SIGINT` by setting up a signal handler and then entering an infinite loop. When the program receives an interrupt signal, the handler function, a pointer to which is supplied to `sigaction()`, executes and exits the program.

SigAction.c

```
1 #include <stdio.h>
2 #include <signal.h>
3 #include <unistd.h>
4 #include <stdlib.h>
5 #include "Practical.h"
6
7 void InterruptSignalHandler(int signalType); // Interrupt signal handling function
8
9 int main(int argc, char *argv[]) {
10     struct sigaction handler; // Signal handler specification structure
11
```

```

12 // Set InterruptSignalHandler() as handler function
13 handler.sa_handler = InterruptSignalHandler;
14 // Create mask that blocks all signals
15 if (sigfillset(&handler.sa_mask) < 0)
16     DieWithSystemMessage("sigfillset() failed");
17 handler.sa_flags = 0; // No flags
18
19 // Set signal handling for interrupt signal
20 if (sigaction(SIGINT, &handler, 0) < 0)
21     DieWithSystemMessage("sigaction() failed for SIGINT");
22
23 for (;;)
24     pause(); // Suspend program until signal received
25
26 exit(0);
27 }
28
29 void InterruptSignalHandler(int signalType) {
30     puts("Interrupt Received. Exiting program.");
31     exit(1);
32 }

```

SigAction.c

1. **Signal handler function prototype:** line 7
2. **Set up signal handler:** lines 10–21
 - **Assign function to handle signal:** line 13
 - **Fill signal mask:** lines 15–16
 - **Set signal handler for SIGINT:** lines 20–21
3. **Loop forever until SIGINT:** lines 23–24
 pause() suspends the process until a signal is received.
4. **Function to handle signal:** lines 29–32
 InterruptSignalHandler() prints a message and exits the program.

So what happens when a signal that would otherwise be delivered is blocked, say, because another signal is being handled? Delivery is postponed until the handler completes. Such a signal is said to be *pending*. **It is important to realize that signals are *not* queued—a signal is either pending or it is not. If the same signal is delivered more than once while it is being handled, the handler is only executed once more after it completes the original execution.** Consider the case where three SIGINT signals arrive while the signal handler for SIGINT is already executing. The first of the three SIGINT signals is blocked; however, the subsequent two signals are lost. When the SIGINT signal handler function completes, the system executes



the handler only *once* again. We must be prepared to handle this behavior in our applications. To see this in action, modify `InterruptSignalHandler()` in `SigAction.c` as follows:

```
void InterruptSignalHandler(int ignored) {
    printf("Interrupt Received.\n");
    sleep(3);
}
```

The signal handler for `SIGINT` sleeps for three seconds and returns, instead of exiting. Now when you execute the program, hit the interrupt key (Control-C) several times in succession. If you hit the interrupt key more than two times in a row, you still only see two “Interrupt Received” messages. The first interrupt signal invokes `InterruptSignalHandler()`, which sleeps for three seconds. The second interrupt is blocked because `SIGINT` is already being handled. The third and fourth interrupts are lost. Be warned that you will no longer be able to stop your program with a keyboard interrupt. You will need to explicitly send another signal (such as `SIGTERM`) to the process using the **kill** command.

One of the most important aspects of signals relates to the sockets interface. If a signal is delivered while the program is blocked in a socket call (such as a `recv()` or `connect()`), and a handler for that signal has been specified, as soon as the handler completes, the socket call will return `-1` with *errno* set to `EINTR`. Thus, **your programs that catch and handle signals need to be prepared for these erroneous returns from system calls that can block.**

Later in this chapter we encounter the first four signals mentioned above. Here we briefly describe the semantics of `SIGPIPE`. Consider the following scenario: A server (or client) has a connected TCP socket, and the other end abruptly and unexpectedly closes the connection, say, because the program crashed. How does the server find out that the connection is broken? The answer is that it doesn’t, until it tries to send or receive on the socket. If it tries to receive first, the call returns `0`. If it tries to send first, at that point, `SIGPIPE` is delivered. Thus, `SIGPIPE` is delivered *synchronously* and not asynchronously. (Why not just return `-1` from `send()`? See exercise 5 at the end of the chapter.)

This fact is especially significant for servers because the default behavior for `SIGPIPE` is to terminate the program. Thus, servers that don’t change this behavior can be terminated by misbehaving clients. **Servers should always handle `SIGPIPE` so that they can detect the client’s disappearance and reclaim any resources that were in use to service it.**

6.3 Nonblocking I/O

The default behavior of a socket call is to block until the requested action is completed. For example, the `recv()` function in `TCPEchoClient.c` (page 44) does not return until at least one message from the echo server is received. Of course, a process with a blocked function is suspended by the operating system.

A socket call may block for several reasons. Data reception functions (`recv()` and `recvfrom()`) block if data is not available. A `send()` on a TCP socket may block if there is not

sufficient space to buffer the transmitted data (see Section 7.1). Connection-related functions for TCP sockets block until a connection has been established. For example, `accept()` in `TCPEchoServer.c` (page 48) blocks until a client establishes a connection with `connect()`. Long round-trip times, high error rate connections, or a slow (or deceased) server may cause a call to `connect()` to block for a significant amount of time. In all of these cases, the function returns only after the request has been satisfied.

What about a program that has other tasks to perform while waiting for call completion (e.g., update busy cursor or respond to user requests)? These programs may have no time to wait on a blocked system call. What about lost UDP datagrams? In `UDPEchoClient.c` (page 54), the client sends a datagram to the server and then waits to receive a response. **If either the datagram sent from the client or the echoed datagram from the server is lost, our echo client blocks indefinitely.** In this case, we need `recvfrom()` to unblock after some amount of time to allow the client to handle the datagram loss. Fortunately, several mechanisms are available for controlling unwanted blocking behaviors. We deal with three such solutions here: nonblocking sockets, asynchronous I/O, and timeouts.

6.3.1 Nonblocking Sockets

One obvious solution to the problem of undesirable blocking is to change the behavior of the socket so that *all* calls are *nonblocking*. For such a socket, if a requested operation can be completed immediately, the call's return value indicates success; otherwise it indicates failure (usually `-1`). In either case the call does not block indefinitely. In the case of failure, we need the ability to distinguish between failure due to blocking and other types of failures. If the failure occurred because the call would have blocked, the system sets *errno* to `EWOULDBLOCK`,² except for `connect()`, which returns an *errno* of `EINPROGRESS`.

We can change the default blocking behavior with a call to `fcntl()` ("file control").

```
int fcntl(int socket, int command, ...)
```

As the name suggests, this call can be used with any kind of file: *socket* must be a valid file (or socket) descriptor. The operation to be performed is given by *command*, which is *always* a system-defined constant. The behavior we want to modify is controlled by flags (*not* the same as socket options) associated with the descriptor, which we can get and set with the `F_GETFL` and `F_SETFL` commands. When setting the socket flags, we must specify the new flags in a variable-length argument list. The flag that controls nonblocking behavior is `O_NONBLOCK`. When getting the socket flags, the variable-length argument list is empty. We demonstrate the use of a nonblocking socket in the next section, where we describe asynchronous I/O in `UDPEchoServer-SIGIO.c` (page 109).

²Some sockets implementations return `EAGAIN`. On many systems, `EAGAIN` and `EWOULDBLOCK` are the same error number.

There are a few exceptions to this model of nonblocking sockets. For UDP sockets, there are no send buffers, so `send()` and `sendto()` never return `EWOULDBLOCK`. For all but the `connect()` socket call, the requested operation either completes before returning or none of the operation is performed. For example, `recv()` either receives data from the socket or returns an error. A nonblocking `connect()` is different. For UDP, `connect()` simply assigns a destination address for future data transmissions so that it never blocks. For TCP, `connect()` initiates the TCP connection setup. If the connection cannot be completed without blocking, `connect()` returns an error, setting `errno` to `EINPROGRESS`, indicating that the socket is still working on making the TCP connection. Of course, subsequent data sends and receives cannot happen until the connection is established. Determining when the connection is complete is beyond the scope of this text,³ so we recommend not setting the socket to nonblocking until after the call to `connect()`.

For eliminating blocking during individual send and receive operations, an alternative is available on some platforms. The `flags` parameter of `send()`, `recv()`, `sendto()`, and `recvfrom()` allows for modification of some aspects of the behavior on a particular call. Some implementations support the `MSG_DONTWAIT` flag, which causes nonblocking behavior in any call where it is set in `flags`.

6.3.2 Asynchronous I/O

The difficulty with nonblocking socket calls is that there is no way of knowing when one would succeed, except by periodically trying it until it does (a process known as “polling”). Why not have the operating system inform the program when a socket call will be successful? That way the program can spend its time doing other work until notified that the socket is ready for something to happen. This is called *asynchronous I/O*, and it works by having the SIGIO signal delivered to the process when some I/O-related event occurs on the socket.

Arranging for SIGIO involves three steps. First, we inform the system of the desired disposition of the signal using `sigaction()`. Then we ensure that signals related to the socket will be delivered to *this* process (because multiple processes can have access to the same socket, there might be ambiguity about which should get it) by making it the owner of the socket, using `fcntl()`. Finally, we mark the socket as being primed for asynchronous I/O by setting a flag (`FASYNC`), again via `fcntl()`.

In our next example, we adapt `UDPEchoServer.c` (page 57) to use asynchronous I/O with nonblocking socket calls. The modified server is able to perform other tasks when there are no clients needing an echo. After creating and binding the socket, instead of calling `recvfrom()` and blocking until a datagram arrives, the asynchronous echo server establishes a signal handler for SIGIO and begins doing other work. When a datagram arrives, the SIGIO signal is delivered to the process, triggering execution of the handler function. The handler function calls `recvfrom()`,

³Well, mostly. Connection completion can be detected using the `select()` call, described in Section 6.5.

echoes back any received datagrams, and then returns, whereupon the main program continues whatever it was doing. Our description details only the code that differs from the original UDP echo server.

UDPEchoServer-SIGIO.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <fcntl.h>
6  #include <sys/file.h>
7  #include <signal.h>
8  #include <errno.h>
9  #include <sys/types.h>
10 #include <sys/socket.h>
11 #include <netdb.h>
12 #include "Practical.h"
13
14 void UseIdleTime();           // Execution during idle time
15 void SIGIOHandler(int signalType); // Handle SIGIO
16
17 int servSock; // Socket -- GLOBAL for signal handler
18
19 int main(int argc, char *argv[]) {
20
21     if (argc != 2) // Test for correct number of arguments
22         DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
23
24     char *service = argv[1]; // First arg: local port
25
26     // Construct the server address structure
27     struct addrinfo addrCriteria;           // Criteria for address
28     memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
29     addrCriteria.ai_family = AF_UNSPEC;     // Any address family
30     addrCriteria.ai_flags = AI_PASSIVE;     // Accept on any address/port
31     addrCriteria.ai_socktype = SOCK_DGRAM;  // Only datagram sockets
32     addrCriteria.ai_protocol = IPPROTO_UDP; // Only UDP protocol
33
34     struct addrinfo *servAddr; // List of server addresses
35     int rtnVal = getaddrinfo(NULL, service, &addrCriteria, &servAddr);
36     if (rtnVal != 0)
37         DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));

```

```

38
39 // Create socket for incoming connections
40 servSock = socket(servAddr->ai_family, servAddr->ai_socktype,
41     servAddr->ai_protocol);
42 if (servSock < 0)
43     DieWithSystemMessage("socket() failed");
44
45 // Bind to the local address
46 if (bind(servSock, servAddr->ai_addr, servAddr->ai_addrlen) < 0)
47     DieWithSystemMessage("bind() failed");
48
49 // Free address list allocated by getaddrinfo()
50 freeaddrinfo(servAddr);
51
52 struct sigaction handler;
53 handler.sa_handler = SIGIOHandler; // Set signal handler for SIGIO
54 // Create mask that mask all signals
55 if (sigfillset(&handler.sa_mask) < 0)
56     DieWithSystemMessage("sigfillset() failed");
57 handler.sa_flags = 0; // No flags
58
59 if (sigaction(SIGIO, &handler, 0) < 0)
60     DieWithSystemMessage("sigaction() failed for SIGIO");
61
62 // We must own the socket to receive the SIGIO message
63 if (fcntl(servSock, F_SETOWN, getpid()) < 0)
64     DieWithSystemMessage("Unable to set process owner to us");
65
66 // Arrange for nonblocking I/O and SIGIO delivery
67 if (fcntl(servSock, F_SETFL, O_NONBLOCK | FASYNC) < 0)
68     DieWithSystemMessage(
69         "Unable to put client sock into non-blocking/async mode");
70
71 // Go off and do real work; echoing happens in the background
72
73 for (;;)
74     UseIdleTime();
75 // NOT REACHED
76 }
77
78 void UseIdleTime() {
79     puts(".");
80     sleep(3); // 3 seconds of activity
81 }
82


```

```

83 void SIGIOHandler(int signalType) {
84     ssize_t numBytesRcvd;
85     do { // As long as there is input...
86         struct sockaddr_storage clntAddr; // Address of datagram source
87         size_t clntLen = sizeof(clntAddr); // Address length in-out parameter
88         char buffer[MAXSTRINGLENGTH]; // Datagram buffer
89
90         numBytesRcvd = recvfrom(servSock, buffer, MAXSTRINGLENGTH, 0,
91                                (struct sockaddr *) &clntAddr, &clntLen);
92         if (numBytesRcvd < 0) {
93             // Only acceptable error: recvfrom() would have blocked
94             if (errno != EWOULDBLOCK)
95                 DieWithSystemMessage("recvfrom() failed");
96         } else {
97             fprintf(stdout, "Handling client ");
98             PrintSocketAddress((struct sockaddr *) &clntAddr, stdout);
99             fputc('\n', stdout);
100
101             ssize_t numBytesSent = sendto(servSock, buffer, numBytesRcvd, 0,
102                                           (struct sockaddr *) &clntAddr, sizeof(clntAddr));
103             if (numBytesSent < 0)
104                 DieWithSystemMessage("sendto() failed");
105             else if (numBytesSent != numBytesRcvd)
106                 DieWithUserMessage("sendto()", "sent unexpected number of bytes");
107         }
108     } while (numBytesRcvd >= 0);
109     // Nothing left to receive
110 }

```

UDPEchoServer-SIGIO.c

1. **Program setup and parameter parsing:** lines 1-24
2. **Prototypes for signal and idle time handler:** lines 14-15
 UseIdleTime() simulates the other tasks of the UDP echo server. SIGIOHandler() handles SIGIO signals. **Note well:** UseIdleTime() **must be prepared for any “slow” system calls—such as reading from a terminal device—to return -1 as a result of the SIGIO signal being delivered and handled** (in which case it should simply verify that *errno* is EINTR and resume whatever it was doing). 
3. **Server socket descriptor:** line 17
 We give the socket descriptor a global scope so that it can be accessed by the SIGIO handler function.
4. **Set up signal handling:** lines 52-69

handler is the **sigaction** structure that describes our desired signal-handling behavior. We fill it in, giving the address of the handling routine and the set of signals we want blocked.

- **Fill in the pointer to the desired handler:** line 53
- **Specify signals to be blocked during handling:** lines 55–56
- **Specify how to handle the SIGIO signal:** lines 59–60
- **Arrange for SIGIO to go to this process:** lines 63–64

The `F_SETOWN` command identifies the process to receive SIGIO for this socket.

- **Set flags for nonblocking and asynchronous I/O:** lines 67–69
- Finally, we mark the socket (with the `FASYNC` flag⁴) to indicate that asynchronous I/O is in use, so SIGIO will be delivered on packet arrival. (Everything up to this point was just saying *how* to deal with SIGIO.) Because we do not want `SIGIOHandler()` to block in `recvfrom()`, we also set the `O_NONBLOCK` flag.

5. **Run forever using idle time when available:** lines 73–74
6. **Perform nonechoing server tasks:** lines 78–81
7. **Handle asynchronous I/O:** lines 83–110

This code is very similar to the loop in our earlier `UDPEchoServer.c` (page 57). One difference is that here we loop until there are no more pending echo requests to satisfy and then return; this technique enables the main program thread to continue what it was doing.

- **Receive echo request:** lines 90–99
- The first call to `recvfrom()` receives the datagram whose arrival prompted the SIGIO signal. Additional datagrams may arrive during execution of the handler, so the `do/while` loop continues to call `recvfrom()` until no more datagrams remain to be received. Because *sock* is a nonblocking socket, `recvfrom()` then returns `-1` with *errno* set to `EWOULDBLOCK`, terminating the loop and the handler function.
- **Send echo reply:** lines 101–106
- Just as in the original UDP echo server, `sendto()` repeats the message back to the client.

6.3.3 Timeouts

In the previous subsection, we relied on the system to notify our program of the occurrence of an I/O-related event. Sometimes, however, we may actually need to know that some I/O event has *not* happened for a certain time period. For example, we have already mentioned that UDP messages can be lost; in case of such a loss, our UDP echo client (or any other client that uses UDP, for that matter) will never receive a response to its request. Of course, the client

⁴The name may be different (e.g., `O_ASYNC`) on some platforms.

cannot tell directly whether a loss has occurred, so it sets a limit on how long it will wait for a response. For example, the UDP echo client might assume that if the server has not responded to its request within two seconds, the server will never respond. The client's reaction to this two-second *timeout* might be to give up or to try again by resending the request.

The standard method of implementing timeouts is to set an alarm before calling a blocking function.

unsigned int alarm(unsigned int secs)

alarm() starts a timer, which expires after the specified number of seconds (secs); alarm() returns the number of seconds remaining for any previously scheduled alarm (or 0 if no alarm was scheduled). When the timer expires, a SIGALRM signal is sent to the process, and the handler function for SIGALRM, if any, is executed.

The code we showed earlier in UDPEchoClient.c (page 54) has a problem if either the echo request or the response is lost: The client blocks indefinitely on recvfrom(), waiting for a datagram that will never arrive. Our next example program, UDPEchoClient-Timeout.c, modifies the original UDP echo client to retransmit the request message if a response from the echo server is not received within a time limit of two seconds. To implement this, the new client installs a handler for SIGALRM, and just before calling recvfrom(), it sets an alarm for two seconds. At the end of that interval of time, the SIGALRM signal is delivered, and the handler is invoked. When the handler returns, the blocked recvfrom() returns -1 with *errno* equal to EINTR. The client then resends the echo request to the server. This timeout and retransmission of the echo request happens up to five times before the client gives up and reports failure. Our program description only details the code that differs from the original UDP echo client.

UDPEchoClient-Timeout.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <errno.h>
6  #include <signal.h>
7  #include <sys/socket.h>
8  #include <netinet/in.h>
9  #include <netdb.h>
10 #include "Practical.h"
11
12 static const unsigned int TIMEOUT_SECS = 2; // Seconds between retransmits
13 static const unsigned int MAXTRIES = 5;    // Tries before giving up

```

```

14
15 unsigned int tries = 0; // Count of times sent - GLOBAL for signal-handler access
16
17 void CatchAlarm(int ignored); // Handler for SIGALRM
18
19 int main(int argc, char *argv[]) {
20
21     if ((argc < 3) || (argc > 4)) // Test for correct number of arguments
22         DieWithUserMessage("Parameter(s)",
23             "<Server Address/Name> <Echo Word> [<Server Port/Service>]\n");
24
25     char *server = argv[1]; // First arg: server address/name
26     char *echoString = argv[2]; // Second arg: word to echo
27
28     size_t echoStringLen = strlen(echoString);
29     if (echoStringLen > MAXSTRINGLENGTH)
30         DieWithUserMessage(echoString, "too long");
31
32     char *service = (argc == 4) ? argv[3] : "echo";
33
34     // Tell the system what kind(s) of address info we want
35     struct addrinfo addrCriteria; // Criteria for address
36     memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
37     addrCriteria.ai_family = AF_UNSPEC; // Any address family
38     addrCriteria.ai_socktype = SOCK_DGRAM; // Only datagram sockets
39     addrCriteria.ai_protocol = IPPROTO_UDP; // Only UDP protocol
40
41     // Get address(es)
42     struct addrinfo *servAddr; // Holder for returned list of server addrs
43     int rtnVal = getaddrinfo(server, service, &addrCriteria, &servAddr);
44     if (rtnVal != 0)
45         DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
46
47     // Create a reliable, stream socket using TCP
48     int sock = socket(servAddr->ai_family, servAddr->ai_socktype,
49         servAddr->ai_protocol); // Socket descriptor for client
50     if (sock < 0)
51         DieWithSystemMessage("socket() failed");
52
53     // Set signal handler for alarm signal
54     struct sigaction handler; // Signal handler
55     handler.sa_handler = CatchAlarm;
56     if (sigfillset(&handler.sa_mask) < 0) // Block everything in handler
57         DieWithSystemMessage("sigfillset() failed");
58     handler.sa_flags = 0;

```



```

59
60 if (sigaction(SIGALRM, &handler, 0) < 0)
61     DieWithSystemMessage("sigaction() failed for SIGALRM");
62
63 // Send the string to the server
64 ssize_t numBytes = sendto(sock, echoString, echoStringLen, 0,
65     servAddr->ai_addr, servAddr->ai_addrlen);
66 if (numBytes < 0)
67     DieWithSystemMessage("sendto() failed");
68 else if (numBytes != echoStringLen)
69     DieWithUserMessage("sendto() error", "sent unexpected number of bytes");
70
71 // Receive a response
72
73 struct sockaddr_storage fromAddr; // Source address of server
74 // Set length of from address structure (in-out parameter)
75 socklen_t fromAddrLen = sizeof(fromAddr);
76 alarm(TIMEOUT_SECS); // Set the timeout
77 char buffer[MAXSTRINGLENGTH + 1]; // I/O buffer
78 while ((numBytes = recvfrom(sock, buffer, MAXSTRINGLENGTH, 0,
79     (struct sockaddr *) &fromAddr, &fromAddrLen)) < 0) {
80     if (errno == EINTR) { // Alarm went off
81         if (tries < MAXTRIES) { // Incremented by signal handler
82             numBytes = sendto(sock, echoString, echoStringLen, 0,
83                 (struct sockaddr *) servAddr->ai_addr, servAddr->ai_addrlen);
84             if (numBytes < 0)
85                 DieWithSystemMessage("sendto() failed");
86             else if (numBytes != echoStringLen)
87                 DieWithUserMessage("sendto() error", "sent unexpected number of bytes");
88         } else
89             DieWithUserMessage("No Response", "unable to communicate with server");
90     } else
91         DieWithSystemMessage("recvfrom() failed");
92 }
93
94 // recvfrom() got something -- cancel the timeout
95 alarm(0);
96
97 buffer[echoStringLen] = '\0'; // Null-terminate the received data
98 printf("Received: %s\n", buffer); // Print the received data
99
100 close(sock);
101 exit(0);
102 }
103

```

```

104 // Handler for SIGALRM
105 void CatchAlarm(int ignored) {
106     tries += 1;
107 }


```

UDPEchoClient-Timeout.c

1. **Program setup and parameter parsing:** lines 1-32
2. **Timeout setup:** lines 12-17
tries is a global variable so that it can be accessed in the signal handler.
3. **Establish signal handler for SIGALRM:** lines 53-61
 This is similar to what we did for SIGIO in UDPEchoServer-SIGIO.c.
4. **Start the alarm timer:** line 76
 When/if the alarm timer expires, the handler `CatchAlarm()` will be invoked.
5. **Retransmission loop:** lines 78-92
 We have to loop here because the SIGALRM will cause the `recvfrom()` to return `-1`. When that happens, we decide whether or not it was a timeout and, if so, retransmit.
 - **Attempt reception:** lines 78-79
 - **Discover the reason for `recvfrom()` failure:** lines 80-91
 If *errno* equals `EINTR`, `recvfrom()` returned because it was interrupted by the SIGALRM while waiting for datagram arrival and not because we got a packet. In this case we assume either the echo request or reply is lost. If we have not exceeded the maximum number of retransmission attempts, we retransmit the request to the server; otherwise, we report a failure. After retransmission, we reset the alarm timer to wake us again if the timeout expires.
6. **Handle echo response reception:** lines 95-98
 - **Cancel the alarm timer:** line 95
 - **Ensure that message is null-terminated:** line 97
`printf()` will output bytes until it encounters a null byte, so we need to make sure one is present (otherwise, our program may crash).
 - **Print the received message:** line 98

6.4 Multitasking

Our TCP echo server handles one client at a time. If additional clients connect while one is already being serviced, their connections will be established and they will be able to send their requests, but the server will not echo back their data until it has finished with the first client. This type of socket application is called an *iterative server*. **Iterative servers work best for**

applications where each client requires a small, bounded amount of work by the server. However, if the time required to handle a client can be long, the overall connection time experienced by any waiting clients may become unacceptably long. To demonstrate the problem, add a `sleep()` after the `connect()` statement in `TCPEchoClient.c` (page 44) and experiment with several clients simultaneously accessing the TCP echo server. (Here, `sleep()` simulates an operation that takes significant time such as slow file or network I/O.) 

Modern operating systems provide a solution to this dilemma. Using constructs like processes or threads, we can farm out responsibility for each client to an independently executing copy of the server. In this section, we will explore several models of such *concurrent servers*, including per-client processes, per-client threads, and constrained multitasking.

6.4.1 Per-Client Processes

Processes are independently executing programs on the same host. In a per-client process server, for each client connection request we simply create a new process to handle the communication. Processes share the resources of the server host, each servicing its client concurrently.

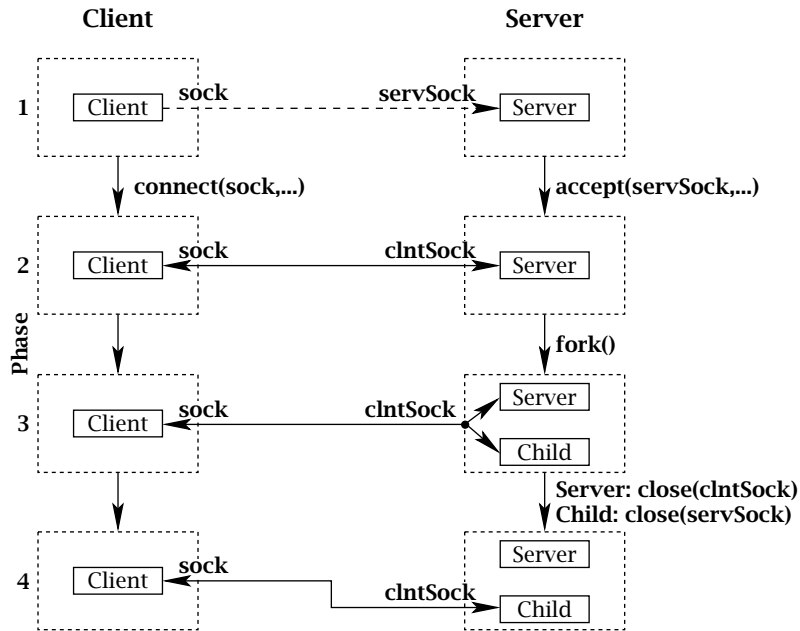
In UNIX, `fork()` attempts the creation of a new process, returning `-1` on failure. On success, a new process is created that is identical to the calling process, except for its process ID and the return value it receives from `fork()`. The two processes thereafter execute independently. The process invoking `fork()` is called the *parent* process, and the newly created process is called the *child*. Since the processes are identical, how do the processes know whether they are parent or child? If the return from `fork()` is `0`, the process knows that it is the child. To the parent, `fork()` returns the process ID of the new child process.

When a child process terminates, it does not automatically disappear. In UNIX parlance, the child becomes a *zombie*. Zombies consume system resources until they are “harvested” by their parent with a call to `waitpid()`, as demonstrated in our next example program, `TCPEchoServer-Fork.c`.

We demonstrate this per-client process, multitasking approach by adapting its use for the TCP echo server. The majority of the program is identical to the original `TCPEchoServer.c` (page 48). The main difference is that the multitasking server creates a new copy of itself each time it accepts a new connection; each copy handles one client and then terminates. No changes are required to `TCPEchoClient.c` (page 44) to work with this new server.

We have decomposed this new echo server to improve readability and to allow reuse in our later examples. Our program commentary is limited to differences between the new server and `TCPEchoServer.c` (page 48).

Figure 6.1 depicts the phases involved in connection setup between the server and a client. The server runs forever, listening for connections on a specified port, and repeatedly (1) accepts an incoming connection from a client and then (2) creates a new process to handle that connection. Note that only the original server process calls `fork()`.

**Figure 6.1:** Forking TCP echo server.**TCPEchoServer-Fork.c**

```

1  #include <stdio.h>
2  #include <unistd.h>
3  #include <stdlib.h>
4  #include <sys/wait.h>
5  #include "Practical.h"
6
7  int main(int argc, char *argv[]) {
8
9      if (argc != 2) // Test for correct number of arguments
10         DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
11
12     char *service = argv[1]; // First arg: local port/service
13     int servSock = SetupTCPServerSocket(service);
14     if (servSock < 0)
15         DieWithUserMessage("SetupTCPServerSocket() failed", "unable to establish");
16
17     unsigned int childProcCount = 0; // Number of child processes
18     for (;;) { // Run forever

```

```

19 // New connection creates a client socket
20 int clntSock = AcceptTCPConnection(servSock);
21 // Fork child process and report any errors
22 pid_t processID = fork();
23 if (processID < 0)
24     DieWithSystemMessage("fork() failed");
25 else if (processID == 0) { // If this is the child process
26     close(servSock); // Child closes parent socket
27     HandleTCPClient(clntSock);
28     exit(0); // Child process terminates
29 }
30
31 printf("with child process: %d\n", processID);
32 close(clntSock); // Parent closes child socket descriptor
33 childProcCount++; // Increment number of child processes
34
35 while (childProcCount) { // Clean up all zombies
36     processID = waitpid((pid_t) - 1, NULL, WNOHANG); // Non-blocking wait
37     if (processID < 0) // waitpid() error?
38         DieWithSystemMessage("waitpid() failed");
39     else if (processID == 0) // No zombie to wait on
40         break;
41     else
42         childProcCount--; // Cleaned up after a child
43 }
44 }
45 // NOT REACHED
46 }

```

TCPEchoServer-Fork.c

1. **Additional include file for waitpid():** line 4
2. **Create server socket:** lines 13–15
SetupTCPServerSocket() allocates, binds, and marks the server socket as ready to accept incoming connections.
3. **Set up to handle multiple processes:** line 17
childProcCount is a variable to count the number of processes.
4. **Process dispatch loop:** lines 18–44
The parent process runs forever, forking a process for each connection request.
 - **Get the next connection:** line 20
AcceptTCPConnection() blocks until a valid connection is established and returns the socket descriptor for that connection. Connection establishment is depicted in the transition from Phase 1 to 2 in Figure 6.1.

- **Create a child process to handle the new connection:** line 22

`fork()` attempts to duplicate the calling process. If the attempt fails, `fork()` returns `-1`. If it succeeds, the child process receives a return value of `0`, and the parent receives a return value of the process ID of the child process. When `fork()` creates a new child process, it copies the socket descriptors from the parent to the child; therefore, after `fork()`, both the parent and child processes have socket descriptors for the listening socket (*servSock*) and the newly created and connected client socket (*clntSock*), as shown in Phase 3 of Figure 6.1.

- **Child process execution:** lines 26–28

The child process is only responsible for dealing with the new client so it can close the listening socket descriptor. However, since the parent process still has a descriptor for the listening socket, this close does not deallocate the socket. This is depicted in the transition from Phase 3 to 4 in Figure 6.1. It is important to note that calling `close()` only terminates the specified socket if no other processes have a reference to the socket. The child process then executes `HandleTCPClient()` to handle the connection. After handling the client, the child process calls `close()` to deallocate the client socket. The child process is terminated with the call to `exit()`.

- **Parent execution continues:** lines 31–33

Since the child is handling the new client, the parent can close the socket descriptor of the new connection socket; again, this does not deallocate the socket because the child process also contains a reference.⁵ (See the transition from Phase 3 to 4 in Figure 6.1.) The parent keeps a count of the number of outstanding child processes in *childProcCount*.

- **Handle zombies:** lines 35–43

After each connection request, the parent server process harvests the zombies created by child process termination. The server repeatedly harvests zombies by calling `waitpid()` until no more of them exist. The first parameter to `waitpid()` (`-1`) is a wildcard that instructs it to take any zombie, regardless of its process ID. The second parameter is a placeholder where `waitpid()` returns the state of the zombie. Since we do not care about the state, we specify `NULL`, and no state is returned. Next comes a flag parameter for customizing the behavior of `waitpid()`. `WNOHANG` causes it to return immediately if no zombies are found. `waitpid()` returns one of three value types: failure (returns `-1`), found zombie (returns *pid* of zombie), and no zombie (returns `0`). If `waitpid()` found a zombie, we need to decrement *childProcCount* and, if more unharvested children exist (*childProcCount* $\neq 0$), look for another zombie. If `waitpid()` returns without finding a zombie, the parent process breaks out of the zombie harvesting loop.

⁵Our description of the child and parent execution assumes that the parent executes `close()` on the client socket before the child. However, it is possible for the client to race ahead and execute the `close()` call on *clntSock* before the server. In this case, the server's `close()` performs the actual socket deallocation, but this does not change the behavior of the server from the client's perspective.

There are several other ways to deal with zombies. On some UNIX variants, the default child termination behavior can be changed so that zombie processes are not created (e.g., `SA_NOCLDWAIT` flag to `sigaction()`). We do not use this approach because it is not portable. Another approach is to establish a handler function for the `SIGCHLD` signal. When a child terminates, a `SIGCHLD` signal is delivered to the original process invoking a specified handler function. The handler function uses `waitpid()` to harvest any zombies. Unfortunately, signals may interrupt at any time, including during blocked system calls (see Section 6.2). The proper method for restarting interrupted system calls differs between UNIX variants. In some systems, restarting is the default behavior. On others, the `sa_flags` field of the `sigaction` structure could be set to `SA_RESTART` to ensure interrupted system calls restart. On other systems, the interrupted system calls return `-1` with `errno` set to `EINTR`. In this case, the program must restart the interrupted function. We do not use any of these approaches because they are not portable, and they complicate the program with issues that we are not addressing. We leave it as an exercise for readers to adapt our TCP echo server to use `SIGCHLD` on their systems.

For this and the rest of the server examples in this chapter, we have factored out the code for setting up the server socket. This code can be found in `TCPServerUtility.c`

SetupTCPServerSocket()

```

1  static const int MAXPENDING = 5; // Maximum outstanding connection requests
2
3  int SetupTCPServerSocket(const char *service) {
4      // Construct the server address structure
5      struct addrinfo addrCriteria;           // Criteria for address match
6      memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
7      addrCriteria.ai_family = AF_UNSPEC;     // Any address family
8      addrCriteria.ai_flags = AI_PASSIVE;     // Accept on any address/port
9      addrCriteria.ai_socktype = SOCK_STREAM; // Only stream sockets
10     addrCriteria.ai_protocol = IPPROTO_TCP; // Only TCP protocol
11
12     struct addrinfo *servAddr; // List of server addresses
13     int rtnVal = getaddrinfo(NULL, service, &addrCriteria, &servAddr);
14     if (rtnVal != 0)
15         DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
16
17     int servSock = -1;
18     for (struct addrinfo *addr = servAddr; addr != NULL; addr = addr->ai_next) {
19         // Create a TCP socket
20         servSock = socket(servAddr->ai_family, servAddr->ai_socktype,
21                         servAddr->ai_protocol);
22         if (servSock < 0)
23             continue; // Socket creation failed; try next address
24
25         // Bind to the local address and set socket to list

```

```

26     if ((bind(servSock, servAddr->ai_addr, servAddr->ai_addrlen) == 0) &&
27         (listen(servSock, MAXPENDING) == 0)) {
28         // Print local address of socket
29         struct sockaddr_storage localAddr;
30         socklen_t addrSize = sizeof(localAddr);
31         if (getsockname(servSock, (struct sockaddr *) &localAddr, &addrSize) < 0)
32             DieWithSystemMessage("getsockname() failed");
33         fputs("Binding to ", stdout);
34         PrintSocketAddress((struct sockaddr *) &localAddr, stdout);
35         fputc('\n', stdout);
36         break;          // Bind and list successful
37     }
38
39     close(servSock); // Close and try again
40     servSock = -1;
41 }
42
43 // Free address list allocated by getaddrinfo()
44 freeaddrinfo(servAddr);
45
46 return servSock;
47 }

```

SetupTCPServerSocket()

The code for getting a new client connection calls `accept()` and prints out information on the client's address.

AcceptTCPConnection()

```

1  int AcceptTCPConnection(int servSock) {
2      struct sockaddr_storage clntAddr; // Client address
3      // Set length of client address structure (in-out parameter)
4      socklen_t clntAddrLen = sizeof(clntAddr);
5
6      // Wait for a client to connect
7      int clntSock = accept(servSock, (struct sockaddr *) &clntAddr, &clntAddrLen);
8      if (clntSock < 0)
9          DieWithSystemMessage("accept() failed");
10
11     // clntSock is connected to a client!
12
13     fputs("Handling client ", stdout);
14     PrintSocketAddress((struct sockaddr *) &clntAddr, stdout);
15     fputc('\n', stdout);

```



```

16
17     return clntSock;
18 }

```

AcceptTCPConnection()

After connection establishment, references to the child socket are done exclusively through the socket descriptor (*clntSock* in this example). In our forking TCP echo server, the IP address and port of the client are only known temporarily in `AcceptTCPConnection()`. What if we want to know the IP address and port outside of `AcceptTCPConnection()`? If we didn't want to return that information from `AcceptTCPConnection()`, we could use the `getpeername()` and `getsockname()` functions described in Section 2.4.6.

6.4.2 Per-Client Thread

Forking a new process to handle each client works, but it is expensive. Every time a process is created, the operating system must duplicate the entire state of the parent process including memory, stack, file/socket descriptors, and so on. *Threads* decrease this cost by allowing multitasking within the same process: A newly created thread simply shares the same address space (code and data) with the parent, negating the need to duplicate the parent state.

The next example program, `TCPEchoServer-Thread.c`, demonstrates a thread-per-client multitasking approach for the TCP echo server using POSIX threads⁶ ("PThreads"). The majority of the program is identical to `TCPEchoServer-Fork.c` (page 118). Again, no changes are required to `TCPEchoClient.c` (page 44) to work with this new server. The program comments are limited to code that differs from the forking echo server.

TCPEchoServer-Thread.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <pthread.h>
5  #include "Practical.h"
6
7  void *ThreadMain(void *arg); // Main program of a thread
8
9  // Structure of arguments to pass to client thread
10 struct ThreadArgs {
11     int clntSock; // Socket descriptor for client

```

⁶Other thread packages work in generally the same manner. We selected POSIX threads because a port of POSIX threads exists for most operating systems.

```

12 };
13
14 int main(int argc, char *argv[]) {
15
16     if (argc != 2) // Test for correct number of arguments
17         DieWithUserMessage("Parameter(s)", "<Server Port/Service>");
18
19     char *service = argv[1]; // First arg: local port/service
20     int servSock = SetupTCPServerSocket(service);
21     if (servSock < 0)
22         DieWithUserMessage("SetupTCPServerSocket() failed", "unable to establish");
23     for (;;) { // Run forever
24         int clntSock = AcceptTCPConnection(servSock);
25
26         // Create separate memory for client argument
27         struct ThreadArgs *threadArgs = (struct ThreadArgs *) malloc(
28             sizeof(struct ThreadArgs));
29         if (threadArgs == NULL)
30             DieWithSystemMessage("malloc() failed");
31         threadArgs->clntSock = clntSock;
32
33         // Create client thread
34         pthread_t threadID;
35         int returnValue = pthread_create(&threadID, NULL, ThreadMain, threadArgs);
36         if (returnValue != 0)
37             DieWithUserMessage("pthread_create() failed", strerror(returnValue));
38         printf("with thread %lu\n", (unsigned long int) threadID);
39     }
40     // NOT REACHED
41 }
42
43 void *ThreadMain(void *threadArgs) {
44     // Guarantees that thread resources are deallocated upon return
45     pthread_detach(pthread_self());
46
47     // Extract socket file descriptor from argument
48     int clntSock = ((struct ThreadArgs *) threadArgs)->clntSock;
49     free(threadArgs); // Deallocate memory for argument
50
51     HandleTCPClient(clntSock);
52
53     return (NULL);
54 }

```

1. **Additional include file for POSIX threads:** line 4

2. **Set up for threads:** lines 7–12

`ThreadMain()` is the function for the POSIX thread to execute. `pthread_create()` allows the caller to pass *one* pointer as an argument to the function to be executed in the new thread. The **ThreadArgs** structure contains the “real” list of parameters. The process creating a new thread allocates and populates the structure before calling `pthread_create()`. In this program the thread function only needs a single argument (*clntSock*), so we could have simply passed a pointer to an integer; however, the *ThreadArgs* structure provides a more general framework for thread argument passing.

3. **Population of thread argument structure:** lines 26–31

We only pass the client socket descriptor to the new thread.

4. **Invocation of the new thread:** lines 33–38

5. **Thread execution:** lines 43–54

`ThreadMain` is the function called by `pthread_create()` when it creates a new thread. The required prototype for the function to be executed by a thread is **`void *fcn(void *)`**, a function that takes a single argument of type **`void *`** and returns a **`void *`**.

■ **Thread detach:** line 45

By default, when a thread's main function returns, state is maintained about the function return code until the parent harvests the results. This is very similar to the behavior for processes. `pthread_detach()` allows the thread state to be immediately deallocated upon completion without parent intervention. `pthread_self()` provides the thread ID of the current thread as a parameter to `pthread_detach()`, much in the way that `getpid()` provides a process its process ID.

■ **Extracting the parameters from the ThreadArgs structure:** lines 48–49

The **ThreadArgs** structure for this program only contains the socket descriptor of the socket connected to the client socket by `accept()`. Because the **ThreadArgs** structure is allocated on a per-connection basis, the new thread can deallocate **threadArgs** once the parameter(s) have been extracted.

■ **HandleTCPClient():** line 51

The thread function calls the same `HandleTCPClient()` function that we have been using all along.

■ **Thread return:** line 53

After creation, the parent does not need to communicate with the thread, so the thread can return a NULL pointer.

Because the parent and thread share the same address space (and thus file/socket descriptors), the parent thread and the per-connection thread do not close the connection and listening sockets, respectively, before proceeding, as the parent and child processes did in the forking example. Figure 6.2 illustrates the actions of the threaded TCP echo server. There are a few disadvantages to using threads instead of processes:

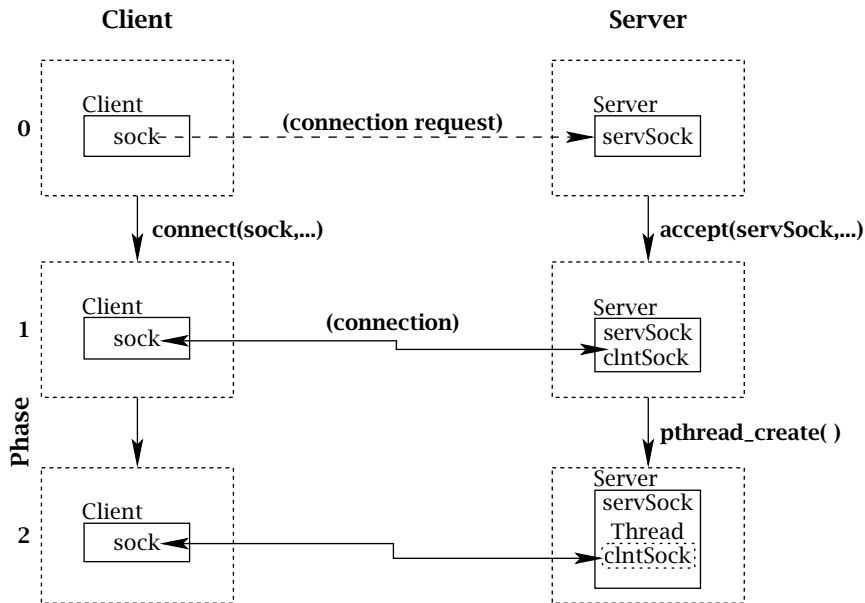


Figure 6.2: Threaded TCP echo server.

- If a child *process* goes awry, it is easy to monitor and kill it from the command line using its process identifier. Threads may not provide this capability on some platforms, so additional server functionality must be provided to monitor and kill individual threads.
- If the operating system is oblivious to the notion of threads, it may give every process the same size time slice. In that case a threaded Web server handling hundreds of clients may get the same amount of CPU time as a game of solitaire.

6.4.3 Constrained Multitasking

Process and thread creation both incur overhead. In addition, the scheduling and context switching among many processes or threads creates extra work for a system. As the number of processes or threads increases, the operating system spends more and more time dealing with this overhead. Eventually, the point is reached where adding another process or thread actually decreases overall performance. That is, a client might actually experience shorter service time if its connection request were queued until some preceding client finished, instead of creating a new process or thread to service it.

We can avoid this problem by limiting the number of processes created by the server, an approach we call *constrained-multitasking servers*. (We present a solution for processes, but it is directly applicable to threads as well.) In this solution, the server begins as the other servers

by creating, binding, and listening to a socket. Then the server creates a set number (say, N) of processes, each of which loops forever, accepting connections from the (same) listening socket. This works because when multiple processes call `accept()` on the same listening socket at the same time, they all block until a connection is established. Then the system picks one process, and the socket descriptor for that new connection is returned *only in that process*; the others remain blocked until the next connection is established, another lucky winner is chosen, and so on.

Our next example program, `TCPEchoServer-ForkN.c`, implements this server model as a modification of the original `TCPEchoServer.c` (page 48), so we only comment on the differences.

TCPEchoServer-ForkN.c

```

1  #include <stdlib.h>
2  #include <unistd.h>
3  #include <stdio.h>
4  #include "Practical.h"
5
6  void ProcessMain(int servSock); // Process main
7
8  int main(int argc, char *argv[]) {
9
10     if (argc != 3) // Test for correct number of arguments
11         DieWithUserMessage("Parameter(s)", "<Server Port/Service> <Process Count>");
12
13     char *service = argv[1]; // First arg: local port
14     unsigned int processLimit = atoi(argv[2]); // Second arg: number of children
15
16     // Server socket
17     int servSock = SetupTCPServerSocket(service);
18
19     // Fork limit-1 child processes
20     for (int processCt = 0; processCt < processLimit - 1; processCt++) {
21         // Fork child process and report any errors
22         pid_t processID = fork();
23         if (processID < 0)
24             DieWithSystemMessage("fork() failed");
25         else if (processID == 0) // If this is the child process
26             ProcessMain(servSock);
27     }
28
29     // Execute last process in parent
30     ProcessMain(servSock);
31     // NOT REACHED
32 }
```

```

33
34 void ProcessMain(int servSock) {
35     for (;;) { // Run forever
36         int clntSock = AcceptTCPConnection(servSock);
37         printf("with child process: %d\n", getpid());
38         HandleTCPClient(clntSock);
39     }
40 }

```

TCPEchoServer-ForkN.c

1. **Prototype for “main” of forked process:** line 6
Each of the N processes executes the `ProcessMain()` function.
2. **Spawning *processLimit* – 1 processes:** lines 19–27
Execute loop *processLimit* – 1 times, each time forking a process that calls `ProcessMain()` with *servSock* as the parameter.
3. **Parent becomes last thread:** line 30
4. **ProcessMain():** lines 34–40
`ProcessMain()` runs forever handling client requests. Effectively, it is the same as the *for(;;)* loop in `TCPEchoServer.c` (page 48).

Because only N processes are created, we limit scheduling overhead, and because each process lives forever handling client requests, we limit process creation overhead. Of course, if we spawn too few processes, we can still have clients waiting unnecessarily for service.

6.5 Multiplexing

Our programs so far have dealt with I/O over a single channel; each version of our echo server deals with only one client connection at a time. However, it is often the case that an application needs the ability to do I/O on multiple channels simultaneously. For example, we might want to provide echo service on several ports at once. The problem with this becomes clear as soon as you consider what happens after the server creates and binds a socket to each port. It is ready to `accept()` connections, but which socket to choose? A call to `accept()` (or `recv()`) on one socket may block, causing established connections to another socket to wait unnecessarily. This problem can be solved using nonblocking sockets, but in that case the server ends up continuously polling the sockets, which is wasteful. We would like to let the server block until *some* socket is ready for I/O.

Fortunately, UNIX provides a way to do this. With the `select()` function, a program can specify a list of descriptors to check for pending I/O; `select()` suspends the program until one of the descriptors in the list becomes ready to perform I/O and returns an indication of

which descriptors are ready. Then the program can proceed with I/O on that descriptor with the assurance that the operation will not block.

```
int select(int maxDescPlus1, fd_set *readDescs, fd_set *writeDescs, fd_set *exceptionDescs,
          struct timeval *timeout)
```

`select()` monitors three separate lists of descriptors. (Note that these descriptors may refer to regular files—such as a terminal input—as well as sockets; we'll see an example of this in our example code later.)

readDescs: Descriptors in this list are checked for immediate input data availability; that is, a call to `recv()` (or `recvfrom()` for a datagram socket) would not block.

writeDescs: Descriptors in this list are checked for the ability to immediately write data; that is, a call to `send()` (or `sendto()` for a datagram socket) would not block.

exceptionDescs: Descriptors in this list are checked for pending exceptions or errors. An example of a pending exception for a TCP socket would be if the remote end of a TCP socket had closed while data were still in the channel; in such a case, the next read or write operation would fail and return `ECONNRESET`.

Passing `NULL` for any of the descriptor vectors makes `select()` ignore that type of I/O. For example, passing `NULL` for `exceptionDescs` causes `select()` to completely ignore exceptions on any sockets. To save space, each of these lists of descriptors is typically represented as a *bit vector*. To include a descriptor in the list, we set the bit in the bit vector corresponding to the number of its descriptor to 1. (For example, `stdin` is descriptor 0, so we would set the first bit in the vector if we want to monitor it.) Programs should not (and need not) rely on knowledge of this implementation strategy, however, because the system provides macros for manipulating instances of the type `fd_set`:

```
void FD_ZERO(fd_set *descriptorVector)
void FD_CLR(int descriptor, fd_set *descriptorVector)
void FD_SET(int descriptor, fd_set *descriptorVector)
int FD_ISSET(int descriptor, fd_set *descriptorVector)
```

`FD_ZERO` empties the list of descriptors. `FD_CLR()` and `FD_SET()` remove and add descriptors to the list, respectively. Membership of a descriptor in a list is tested by `FD_ISSET()`, which returns nonzero if the given descriptor is in the list, and 0 otherwise.

The maximum number of descriptors that can be contained in a list is given by the system-defined constant `FD_SETSIZE`. While this number can be quite large, most applications use very few descriptors. To make the implementation more efficient, the `select()` function allows us

to pass a *hint*, which indicates the largest descriptor number that needs to be considered in any of the lists. In other words, *maxDescPlus1* is the smallest descriptor number that does *not* need to be considered, which is simply the maximum descriptor value plus one. For example, if descriptors 0, 3, and 5 are set in the descriptor list, we would set *maxDescPlus1* to the maximum descriptor value (5) plus one. Notice that *maxDescPlus1* applies for *all three* descriptor lists. If the exception descriptor list's largest descriptor is 7, while the read and write descriptor lists' largest are 5 and 2, respectively, then we set *maxDescPlus1* to 8.

What would you pay for the ability to listen simultaneously to so many descriptors for up to *three* types of I/O? Don't answer yet because *select()* does even more! The last parameter (*timeout*) allows control over how long *select()* will wait for something to happen. The *timeout* is specified with a **timeval** data structure:

```
struct timeval {
    time_t tv_sec;      // Seconds
    time_t tv_usec;     // Microseconds
};
```

If the time specified in the *timeval* structure elapses before any of the specified descriptors becomes ready for I/O, *select()* returns the value 0. If *timeout* is NULL, *select()* has no timeout bound and waits until some descriptor becomes ready. **Setting both *tv_sec* and *tv_usec* to 0 causes *select()* to return immediately, enabling polling of I/O descriptors.**



If no errors occur, *select()* returns the total number of descriptors prepared for I/O. To indicate the descriptors ready for I/O, *select()* changes the descriptor lists so that only the positions corresponding to ready descriptors are set. For example, if descriptors 0, 3, and 5 are set in the initial read descriptor list, the write and exception descriptor lists are NULL, and descriptors 0 and 5 have data available for reading, *select()* returns 2, and only positions 0 and 5 are set in the returned read descriptor list. An error in *select()* is indicated by a return value of -1.

Let's reconsider the problem of running the echo service on multiple ports. If we create a socket for each port, we could list these sockets in a *readDescriptor* list. A call to *select()*, given such a list, would suspend the program until an echo request arrives for at least one of the descriptors. We could then handle the connection setup and echo for that particular socket. Our next example program, *TCPEchoServer-Select.c*, implements this model. The user can specify an arbitrary number of ports to monitor. Notice that a connection request is considered I/O and prepares a socket descriptor for reading by *select()*. To illustrate that *select()* works on nonsocket descriptors as well, this server also watches for input from the standard input stream, which it interprets as a signal to terminate itself.

TCPEchoServer-Select.c

```
1 #include <sys/time.h>
2 #include <stdlib.h>
3 #include <stdio.h>
```



```

4 #include <unistd.h>
5 #include <fcntl.h>
6 #include <stdbool.h>
7 #include "Practical.h"
8
9 int main(int argc, char *argv[]) {
10
11     if (argc < 3) // Test for correct number of arguments
12         DieWithUserMessage("Parameter(s)", "<Timeout (secs.)> <Port/Service1> ...");
13
14     long timeout = atol(argv[1]); // First arg: Timeout
15     int noPorts = argc - 2;      // Number of ports is argument count minus 2
16
17     // Allocate list of sockets for incoming connections
18     int servSock[noPorts];
19     // Initialize maxDescriptor for use by select()
20     int maxDescriptor = -1;
21
22     // Create list of ports and sockets to handle ports
23     for (int port = 0; port < noPorts; port++) {
24         // Create port socket
25         servSock[port] = SetupTCPServerSocket(argv[port + 2]);
26
27         // Determine if new descriptor is the largest
28         if (servSock[port] > maxDescriptor)
29             maxDescriptor = servSock[port];
30     }
31
32     puts("Starting server: Hit return to shutdown");
33     bool running = true; // true if server should continue running
34     fd_set sockSet;      // Set of socket descriptors for select()
35     while (running) {
36         /* Zero socket descriptor vector and set for server sockets
37          * This must be reset every time select() is called */
38         FD_ZERO(&sockSet);
39         // Add keyboard to descriptor vector
40         FD_SET(STDIN_FILENO, &sockSet);
41         for (int port = 0; port < noPorts; port++)
42             FD_SET(servSock[port], &sockSet);
43
44         // Timeout specification; must be reset every time select() is called
45         struct timeval selTimeout; // Timeout for select()
46         selTimeout.tv_sec = timeout; // Set timeout (secs.)
47         selTimeout.tv_usec = 0;      // 0 microseconds
48

```

```

49 // Suspend program until descriptor is ready or timeout
50 if (select(maxDescriptor + 1, &sockSet, NULL, NULL, &selTimeout) == 0)
51     printf("No echo requests for %ld secs...Server still alive\n", timeout);
52 else {
53     if (FD_ISSET(0, &sockSet)) { // Check keyboard
54         puts("Shutting down server");
55         getchar();
56         running = false;
57     }
58
59     // Process connection requests
60     for (int port = 0; port < noPorts; port++)
61         if (FD_ISSET(servSock[port], &sockSet)) {
62             printf("Request on port %d: ", port);
63             HandleTCPClient(AcceptTCPConnection(servSock[port]));
64         }
65     }
66 }
67
68 // Close sockets
69 for (int port = 0; port < noPorts; port++)
70     close(servSock[port]);
71
72 exit(0);
73 }

```

TCPEchoServer-Select.c

1. **Set up a socket for each port:** lines 23–30
We store the socket descriptors in an array, one per argument to the program.
2. **Create list of file descriptors for select():** line 25
3. **Set timer for select():** lines 44–47
4. **select() execution:** lines 49–65
 - **Handle timeout:** line 51
 - **Check keyboard descriptor:** lines 53–57
If the user presses return, descriptor STDIN_FILENO will be ready for reading; in that case the server terminates itself.
 - **Check the socket descriptors:** lines 60–64
Test each descriptor, accepting and handling the valid connections.
5. **Wrap-up:** lines 68–72
Close all ports and exit.

`select()` is a powerful function. It can also be used to implement a timeout version of any of the blocking I/O functions (e.g., `recv()`, `accept()`) without using alarms.

6.6 Multiple Recipients

So far, all of our programs have dealt with communication involving two entities, usually a server and a client. Such one-to-one communication is sometimes called *unicast* because only one (“uni”) copy of the data is sent (“cast”). Sometimes we would like to send the *same* information to more than one recipient. You probably know that computers in an office or home are often connected to a local area network, and sometimes we would like to send information that can be received by every host on the network. For example, a computer that has a printer attached may advertise it for use by other hosts on the network by sending a message this way; the operating systems of other machines receive these advertisements and make the shared resources available to their users. Instead of unicasting the message to every host on the network—which requires us not only to *know* the address of every host on the network, but also to call `sendto()` on the message once for each host—we would like to be able to call `sendto()` just once and have the *network* handle the duplication for us.

Or consider a typical case in which the sender has a single connection to the Internet, such as a cable or DSL modem. Sending the same data to multiple recipients scattered throughout the Internet with unicast requires that the same information be sent over that link many times (Figure 6.3a). If the sender’s first-hop connection has limited outgoing capacity (say, one

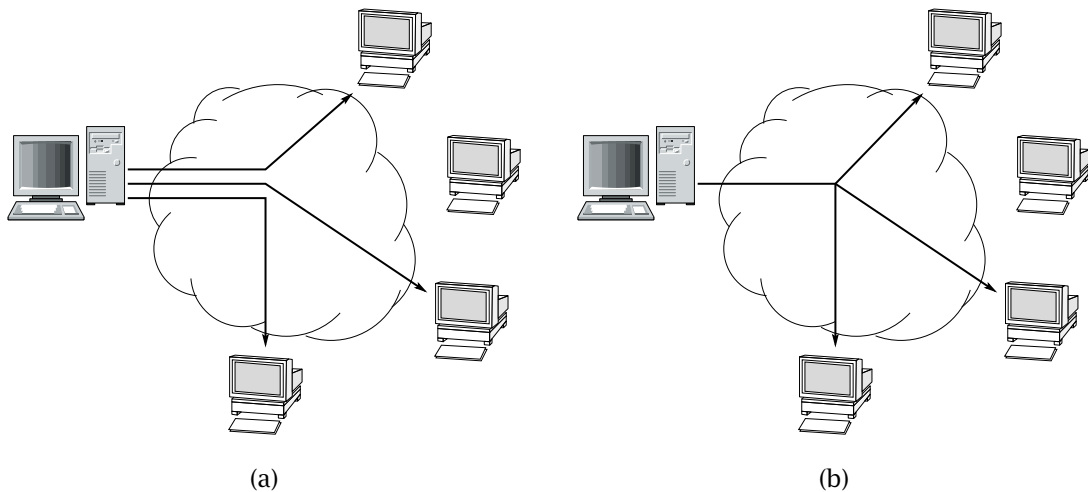


Figure 6.3: (a) Unicasting the same data to multiple recipients; (b) doing the same thing with multicast.

1 Mbps or so), it may not even be *possible* to send some kinds of information—say, a real-time video stream at 1 Mbps—to more than one recipient without exceeding the first-hop link capacity, resulting in many lost packets and poor quality. Clearly, it would be more efficient if the information could be duplicated *after* it crosses the first link, as in Figure 6.3b. This saves bandwidth *and* simplifies life for the sending program.

You may be surprised to learn that the sockets interface over TCP/IP provides access to services like this—albeit with some restrictions. There are two types of network duplication service: *broadcast* and *multicast*. With broadcast, the program calls `sendto()` once, and the message is automatically delivered to *all* hosts on the local network. With multicast, the message is sent once and delivered to a specific (possibly empty) *group* of hosts throughout the Internet—namely, those that have indicated to the network that they should receive messages sent to that group.

We mentioned that there are some restrictions on these services. **The first is that *only UDP sockets can use broadcast and multicast services*. The second is that *broadcast only covers a local scope, typically a local area network*. The third restriction is that *multicast across the entire Internet is presently not supported by most Internet service providers*.** In spite of these restrictions, these services can often be useful. For example, it is often useful to use multicast within a site such as a campus network, or broadcast to local hosts.



6.6.1 Broadcast

UDP datagrams can be sent to all nodes on an attached local network by sending them to a special address. In IPv4 it is called the “limited broadcast address,” and it is the all-ones address (in dotted-quad notation, 255.255.255.255). In IPv6 it is called the “all-nodes address (link scope)” and has the value FF02::1. Routers do not forward packets addressed to either one of these addresses, so neither one will take a datagram beyond the local network to which the sender is connected. Each does, however, deliver the sent packet to *every* node on the network; typically, this is achieved using the hardware broadcast capability of the local network. (Not all links support broadcast; in particular, point-to-point links do not. If none of a host’s interfaces support broadcast, any attempt to use it will result in an error.) Note also that a broadcast UDP datagram will actually be “heard” at a host only if some program on that host is listening for datagrams on the *port* to which the datagram is addressed.

What about a network-wide broadcast address to send a message to all hosts? There is no such address. To see why, consider the impact on the network of a broadcast to every host on the Internet. Sending a single datagram would result in an extremely large number of packet duplications by the routers, and bandwidth would be consumed on each and every network. The consequences of misuse (malicious or accidental) are too great, so the designers of IP left out such an Internet-wide broadcast facility on purpose. Even with these restrictions, broadcast on the local link can be very useful. Often it is used in state exchange for network games where the players are all on the same local area network (e.g., Ethernet).

There is one other difference between a broadcast sender and a regular sender: before sending to the broadcast address, the special socket option `SO_BROADCAST` must be set.

In effect, this asks the system for “permission” to broadcast. We demonstrate the use of UDP broadcast in `BroadcastSender.c`. Our sender broadcasts a given string every three seconds to the limited broadcast address of the address family indicated by the first argument.

BroadcastSender.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <sys/socket.h>
6  #include <arpa/inet.h>
7  #include "Practical.h"
8
9  static const char *IN6ADDR_ALLNODES = "FF02::1"; // v6 addr not built in
10
11 int main(int argc, char *argv[]) {
12
13     if (argc != 4) // Test for correct number of arguments
14         DieWithUserMessage("Parameter(s)", "[4|6] <Port> <String to send>");
15
16     in_port_t port = htons((in_port_t) atoi(argv[2]));
17
18     struct sockaddr_storage destStorage;
19     memset(&destStorage, 0, sizeof(destStorage));
20
21     size_t addrSize = 0;
22     if (argv[1][0] == '4') {
23         struct sockaddr_in *destAddr4 = (struct sockaddr_in *) &destStorage;
24         destAddr4->sin_family = AF_INET;
25         destAddr4->sin_port = port;
26         destAddr4->sin_addr.s_addr = INADDR_BROADCAST;
27         addrSize = sizeof(struct sockaddr_in);
28     } else if (argv[1][0] == '6') {
29         struct sockaddr_in6 *destAddr6 = (struct sockaddr_in6 *) &destStorage;
30         destAddr6->sin6_family = AF_INET6;
31         destAddr6->sin6_port = port;
32         inet_pton(AF_INET6, IN6ADDR_ALLNODES, &destAddr6->sin6_addr);
33         addrSize = sizeof(struct sockaddr_in6);
34     } else {
35         DieWithUserMessage("Unknown address family", argv[1]);
36     }
37
38     struct sockaddr *destAddress = (struct sockaddr *) &destStorage;

```

```

39
40     size_t msgLen = strlen(argv[3]);
41     if (msgLen > MAXSTRINGLENGTH) // Input string fits?
42         DieWithUserMessage("String too long", argv[3]);
43
44     // Create socket for sending/receiving datagrams
45     int sock = socket(destAddress->sa_family, SOCK_DGRAM, IPPROTO_UDP);
46     if (sock < 0)
47         DieWithSystemMessage("socket() failed");
48
49     // Set socket to allow broadcast
50     int broadcastPerm = 1;
51     if (setsockopt(sock, SOL_SOCKET, SO_BROADCAST, &broadcastPerm,
52         sizeof(broadcastPerm)) < 0)
53         DieWithSystemMessage("setsockopt() failed");
54
55     for (;;) { // Run forever
56         // Broadcast msgString in datagram to clients every 3 seconds
57         ssize_t numBytes = sendto(sock, argv[3], msgLen, 0, destAddress, addrSize);
58         if (numBytes < 0)
59             DieWithSystemMessage("sendto() failed");
60         else if (numBytes != msgLen)
61             DieWithUserMessage("sendto()", "sent unexpected number of bytes");
62
63         sleep(3); // Avoid flooding the network
64     }
65     // NOT REACHED
66 }

```

BroadcastSender.c

1. **Declare constant address:** line 9
Somewhat surprisingly, the all-nodes group address is not defined as a named system constant, so we give it a name here.
2. **Parameter processing:** lines 13–16
3. **Destination address storage:** lines 18–19
We use a **sockaddr_storage** structure to hold the destination broadcast address, since it may be either IPv4 or IPv6.
4. **Setting up destination address:** lines 21–38
We set the destination address according to the given type and remember the size for later use. Finally, we save the address in a pointer to a generic **sockaddr**.
5. **Socket creation:** lines 44–47

6. Setting permission to broadcast: lines 49–53

By default, sockets cannot broadcast. Setting the `SO_BROADCAST` option for the socket enables socket broadcast.

7. Repeatedly broadcast: lines 55–64

Send the argument string every three seconds to all hosts on the network.

Note that a receiver program does not need to do anything special to receive broadcast datagrams (except bind to the appropriate port). Writing a program to receive the broadcasts sent out by `BroadcastSender.c` is left as an exercise.

6.6.2 Multicast

For the sender using multicast is very similar to using unicast. The difference is in the form of the address. A multicast address identifies a set of receivers who have “asked” the network to deliver messages sent to that address. (This is the receiver’s responsibility; see below.) A range of the address space is set aside for multicast in both IPv4 and IPv6. IPv4 multicast addresses are in the range 224.0.0.0 to 239.255.255.255. IPv6 multicast addresses are those whose first byte contains `0xFF`, that is, all ones. The IPv6 multicast address space has a fairly complicated structure that is mostly beyond the scope of this book. (The reader is referred to [4] for details.) For our examples we’ll use addresses beginning with `FF1E`; they are valid for transient use in global applications. (The third hex digit “1” indicates a multicast address that is not permanently assigned for any particular purpose, while the fourth digit “E” indicates global scope.) An example would be `FF1E::1234`.

Our example multicast sender, shown in file `MulticastSender.c`, takes a multicast address and port as an argument, and sends a given string to that address and port every three seconds.

MulticastSender.c

```

1  #include <stdlib.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include <netdb.h>
5  #include "Practical.h"
6
7  int main(int argc, char *argv[]) {
8
9      if (argc < 4 || argc > 5) // Test for number of parameters
10         DieWithUserMessage("Parameter(s)",
11                             "<Multicast Address> <Port> <Send String> [<TTL>]");
12
13     char *multicastIPString = argv[1]; // First arg: multicast IP address
14     char *service = argv[2]; // Second arg: multicast port/service
15     char *sendString = argv[3]; // Third arg: string to multicast

```

```

16
17     size_t sendStringLen = strlen(sendString);
18     if (sendStringLen > MAXSTRINGLENGTH) // Check input length
19         DieWithUserMessage("String too long", sendString);
20
21     // Fourth arg (optional): TTL for transmitting multicast packets
22     int multicastTTL = (argc == 5) ? atoi(argv[4]) : 1;
23
24     // Tell the system what kind(s) of address info we want
25     struct addrinfo addrCriteria; // Criteria for address match
26     memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
27     addrCriteria.ai_family = AF_UNSPEC; // v4 or v6 is OK
28     addrCriteria.ai_socktype = SOCK_DGRAM; // Only datagram sockets
29     addrCriteria.ai_protocol = IPPROTO_UDP; // Only UDP please
30     addrCriteria.ai_flags |= AI_NUMERICHOST; // Don't try to resolve address
31
32     struct addrinfo *multicastAddr; // Holder for returned address
33     int rtnVal= getaddrinfo(multicastIPString, service,
34                             &addrCriteria, &multicastAddr);
35     if (rtnVal != 0)
36         DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
37
38     // Create socket for sending datagrams
39     int sock = socket(multicastAddr->ai_family, multicastAddr->ai_socktype,
40                      multicastAddr->ai_protocol);
41     if (sock < 0)
42         DieWithSystemMessage("socket() failed");
43
44     // Set TTL of multicast packet. Unfortunately this requires
45     // address-family-specific code
46     if (multicastAddr->ai_family == AF_INET6) { // v6-specific
47         // The v6 multicast TTL socket option requires that the value be
48         // passed in as an integer
49         if (setsockopt(sock, IPPROTO_IPV6, IPV6_MULTICAST_HOPS,
50                        &multicastTTL, sizeof(multicastTTL)) < 0)
51             DieWithSystemMessage("setsockopt(IPV6_MULTICAST_HOPS) failed");
52     } else if (multicastAddr->ai_family == AF_INET) { // v4 specific
53         // The v4 multicast TTL socket option requires that the value be
54         // passed in as an unsigned char
55         u_char mcTTL = (u_char) multicastTTL;
56         if (setsockopt(sock, IPPROTO_IP, IP_MULTICAST_TTL, &mcTTL,
57                        sizeof(mcTTL)) < 0)
58             DieWithSystemMessage("setsockopt(IP_MULTICAST_TTL) failed");
59     } else {
60         DieWithUserMessage("Unable to set TTL", "invalid address family");

```



```

61  }
62
63  for (;;) { // Run forever
64      // Multicast the string to all who have joined the group
65      ssize_t numBytes = sendto(sock, sendString, sendStringLen, 0,
66                               multicastAddr->ai_addr, multicastAddr->ai_addrlen);
67      if (numBytes < 0)
68          DieWithSystemMessage("sendto() failed");
69      else if (numBytes != sendStringLen)
70          DieWithUserMessage("sendto()", "sent unexpected number of bytes");
71      sleep(3);
72  }
73  // NOT REACHED
74  }

```

MulticastSender.c

Note that unlike the broadcast sender, the multicast sender does not need to set the permission to multicast. On the other hand, the multicast sender may set the *TTL* (“time-to-live”) value for the transmitted datagrams. Every packet contains a counter, which is initialized to some default value when the packet is first sent and decremented by each router that handles the packet. When this counter reaches 0, the packet is discarded. The TTL mechanism (which can be changed by setting a socket option) allows us to control the initial value of this counter and thus limit the number of hops a multicast packet can traverse. For example, by setting *TTL* = 1, the multicast packet will not go beyond the local network.

As mentioned earlier, the multicast network service duplicates and delivers the message only to a specific set of receivers. This set of receivers, called a *multicast group*, is identified by a particular multicast (or group) address. These receivers need some mechanism to notify the network of their interest in receiving data sent to a particular multicast address. Once notified, the network can begin forwarding the multicast messages to the receiver. This notification of the network, called “joining a group,” is accomplished via a multicast request (signaling) message sent (transparently) by the underlying protocol implementation. To cause this to happen, the receiving program needs to invoke an address-family-specific multicast socket option. For IPv4 it is `IP_ADD_MEMBERSHIP`; for IPv6 it is (surprisingly enough) `IPv6_ADD_MEMBERSHIP`. This socket option takes a structure containing the address of the multicast “group” to be joined. Alas, this structure is also different for the two versions:

```

struct ip_mreq {
    struct in_addr imr_multiaddr; // Group address
    struct in_addr imr_interface; // local interface to join on
};

```

The IPv6 version differs only in the type of addresses it contains:

```

struct ipv6_mreq {

```

```

    struct in6_addr ipv6mr_multiaddr; // IPv6 multicast address of group
    unsigned int ipv6mr_interface;    // local interface to join no
};

```

Our multicast receiver contains a fair amount of version-specific code to handle the joining process.

MulticastReceiver.c

```

1  #include <stdlib.h>
2  #include <string.h>
3  #include <unistd.h>
4  #include <netdb.h>
5  #include "Practical.h"
6
7  int main(int argc, char *argv[]) {
8
9      if (argc != 3)
10         DieWithUserMessage("Parameter(s)", "<Multicast Address> <Port>");
11
12     char *multicastAddrString = argv[1]; // First arg: multicast addr (v4 or v6!)
13     char *service = argv[2];           // Second arg: port/service
14
15     struct addrinfo addrCriteria;        // Criteria for address match
16     memset(&addrCriteria, 0, sizeof(addrCriteria)); // Zero out structure
17     addrCriteria.ai_family = AF_UNSPEC;   // v4 or v6 is OK
18     addrCriteria.ai_socktype = SOCK_DGRAM; // Only datagram sockets
19     addrCriteria.ai_protocol = IPPROTO_UDP; // Only UDP protocol
20     addrCriteria.ai_flags |= AI_NUMERICHOST; // Don't try to resolve address
21
22     // Get address information
23     struct addrinfo *multicastAddr;      // List of server addresses
24     int rtnVal = getaddrinfo(multicastAddrString, service,
25                             &addrCriteria, &multicastAddr);
26     if (rtnVal != 0)
27         DieWithUserMessage("getaddrinfo() failed", gai_strerror(rtnVal));
28
29     // Create socket to receive on
30     int sock = socket(multicastAddr->ai_family, multicastAddr->ai_socktype,
31                     multicastAddr->ai_protocol);
32     if (sock < 0)
33         DieWithSystemMessage("socket() failed");
34
35     if (bind(sock, multicastAddr->ai_addr, multicastAddr->ai_addrlen) < 0)
36         DieWithSystemMessage("bind() failed");
37

```

```

38 // Unfortunately we need some address-family-specific pieces
39 if (multicastAddr->ai_family == AF_INET6) {
40     // Now join the multicast "group" (address)
41     struct ipv6_mreq joinRequest;
42     memcpy(&joinRequest.ipv6mr_multiaddr, &((struct sockaddr_in6 *)
43         multicastAddr->ai_addr)->sin6_addr, sizeof(struct in6_addr));
44     joinRequest.ipv6mr_interface = 0; // Let system choose the i/f
45     puts("Joining IPv6 multicast group...");
46     if (setsockopt(sock, IPPROTO_IPV6, IPV6_JOIN_GROUP,
47         &joinRequest, sizeof(joinRequest)) < 0)
48         DieWithSystemMessage("setsockopt(IPV6_JOIN_GROUP) failed");
49 } else if (multicastAddr->ai_family == AF_INET) {
50     // Now join the multicast "group"
51     struct ip_mreq joinRequest;
52     joinRequest.imr_multiaddr =
53         ((struct sockaddr_in *) multicastAddr->ai_addr)->sin_addr;
54     joinRequest.imr_interface.s_addr = 0; // Let the system choose the i/f
55     printf("Joining IPv4 multicast group...\n");
56     if (setsockopt(sock, IPPROTO_IP, IP_ADD_MEMBERSHIP,
57         &joinRequest, sizeof(joinRequest)) < 0)
58         DieWithSystemMessage("setsockopt(IPV4_ADD_MEMBERSHIP) failed");
59 } else {
60     DieWithSystemMessage("Unknown address family");
61 }
62 // Free address structure(s) allocated by getaddrinfo()
63 freeaddrinfo(multicastAddr);
64
65 char recvString[MAXSTRINGLENGTH + 1]; // Buffer to receive into
66 // Receive a single datagram from the server
67 int recvStringLength = recvfrom(sock, recvString, MAXSTRINGLENGTH, 0, NULL, 0);
68 if (recvStringLength < 0)
69     DieWithSystemMessage("recvfrom() failed");
70
71 recvString[recvStringLength] = '\0'; // Terminate the received string
72 // Note: sender did not send the terminal 0
73 printf("Received: %s\n", recvString);
74
75 close(sock);
76 exit(0);
77 }

```

MulticastReceiver.c

The multicast receiver joins the group, waits to receive a message, prints it, and then exits.

6.6.3 Broadcast vs. Multicast

The decision of using broadcast or multicast in an application depends on several issues, including the fraction of network hosts interested in receiving the data, and the knowledge of the communicating parties. Broadcast works well if a large percentage of the network hosts wish to receive the message; however, if few hosts need to receive the packet, broadcast “imposes on” all hosts in the network for the benefit of a few. Multicast is preferred because it limits the duplication of data to those that have expressed interest. The disadvantages of multicast are (1) it is presently not supported globally, and (2) the sender and receiver must agree on an IP multicast address in advance. Knowledge of an address is not required for broadcast. In some contexts (local), this makes broadcast a better mechanism for discovery than multicast. All hosts can receive broadcast by default, so it is simple to ask all hosts a question like “Where’s the printer?” On the other hand, for wide-area applications, multicast is the only choice.

Exercises

1. State precisely the conditions under which an iterative server is preferable to a multiprocess server.
2. Would you ever need to implement a timeout in a client or server that uses TCP?
3. Why do we make the server socket nonblocking in `UDPEchoServer-SIGIO.c`? In particular, what bad thing might happen if we did not?
4. How can you determine the minimum and maximum allowable sizes for a socket’s send and receive buffers? Determine the minimums for your system.
5. This exercise considers the reasoning behind the `SIGPIPE` mechanism a little further. Recall that `SIGPIPE` is delivered when a program tries to send on a TCP socket whose connection has gone away in the meantime. An alternative approach would be to simply have the `send()` fail with `ECONNRESET`. Why might the signal-based approach be better than conveying this information by return value?
6. What do you think will happen if you use the program in `MulticastReceiver.c` while the program `BroadcastSender.c` is running on a host connected to the same LAN?