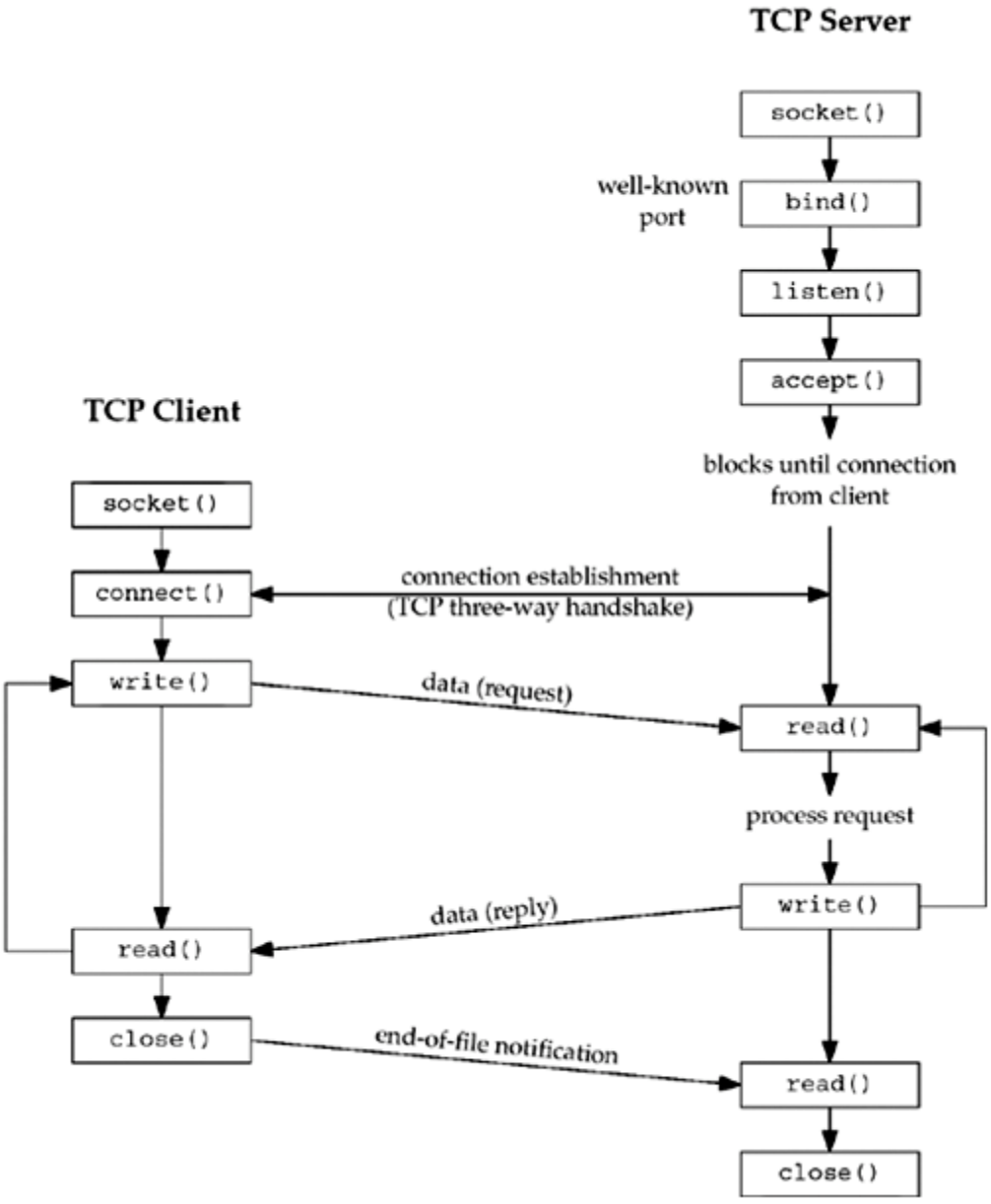
Unit - 5

Elementary TCP Sockets

The elementary socket functions required to write a complete TCP client and server, along with concurrent servers, a common Unix technique for providing concurrency when numerous clients are connected to the same server at the same time. Each client connection causes the server to fork a new process just for that client. In this chapter, we consider only the one-process-per-client model using fork.

The figure below shows a timeline of the typical scenario that takes place between a TCP client and server. First, the server is started, then sometime later, a client is started that connects to the server. We assume that the client sends a request to the server, the server processes the request, and the server sends a reply back to the client. This continues until the client closes its end of the connection, which sends an end-of-file notification to the server. The server then closes its end of the connection and either terminates or waits for a new client connection.



socket Function

To perform network I/O, the first thing a process must do is call the socket function, specifying the type of communication protocol desired (TCP using IPv4, UDP using IPv6, Unix domain stream protocol, etc.).

```
#include <sys/socket.h>

int socket (int family, int type, int protocol);

/* Returns: non-negative descriptor if OK, -1 on error */
```

Arguments:

- *family* specifies the protocol family and is one of the constants in the table below. This argument is often referred to as *domain* instead of *family*.

family	**Description**
AF_INET	IPv4 protocols
AF_INET6	IPv6 protocols
AF_LOCAL	Unix domain protocols
AF_ROUTE	Routing sockets
AF_KEY	Key socket

- The socket *type* is one of the constants shown in table below:

type	**Description**
SOCK_STREAM	stream socket
SOCK_DGRAM	datagram socket
SOCK_SEQPACKET	sequenced packet socket

type	Description
SOCK_RAW	raw socket

- The *protocol* argument to the socket function should be set to the specific protocol type found in the table below, or 0 to select the system's default for the given combination of *family* and *type*.

protocol	Description
IPPROTO_TCP	TCP transport protocol
IPPROTO_UDP	UDP transport protocol
IPPROTO_SCTP	SCTP transport protocol

Not all combinations of socket *family* and *type* are valid. The table below shows the valid combinations, along with the actual protocols that are valid for each pair. The boxes marked "Yes" are valid but do not have handy acronyms. The blank boxes are not supported.

	AF_INET	AF_INET6	AF_LOCAL	AF_ROUTE
SOCK_STREAM	TCP/SCTP	TCP/SCTP	Yes	
SOCK_DGRAM	UDP	UDP	Yes	
SOCK_SEQPACKET	SCTP	SCTP	Yes	
SOCK_RAW	IPv4	IPv6		Yes

Notes:

- You may also encounter the corresponding PF_*xxx* constant as the first argument to socket. This is discussed in the next section in this Chapter.
- You may encounter AF_UNIX (the historical Unix name) instead of AF_LOCAL (the POSIX name).
- Linux supports a new socket type, SOCK_PACKET, that provides access to the datalink, similar to BPF and DLPI.

- The key socket, AF_KEY, is newer than the others. It provides support for cryptographic security. Similar to the way that a routing socket (AF_ROUTE) is an interface to the kernel's routing table, the key socket is an interface into the kernel's key table.

On success, the socket function returns a small non-negative integer value, similar to a file descriptor. We call this a **socket descriptor**, or a *sockfd*. To obtain this socket descriptor, all we have specified is a protocol family (IPv4, IPv6, or Unix) and the socket type (stream, datagram, or raw). We have not yet specified either the local protocol address or the foreign protocol address.

AF_*xxx* Versus PF_*xxx*

The "AF_" prefix stands for "address family" and the "PF_" prefix stands for "protocol family." Historically, the intent was that a single protocol family might support multiple address families and that the PF_ value was used to create the socket and the AF_ value was used in socket address structures. But in actuality, a protocol family supporting multiple address families has never been supported and the <sys/socket.h> header defines the PF_ value for a given protocol to be equal to the AF_ value for that protocol. While there is no guarantee that this equality between the two will always be true, should anyone change this for existing protocols, lots of existing code would break.

To conform to existing coding practice, we use only the AF_ constants in this text, although you may encounter the PF_ value, mainly in calls to socket.

[p98-99]

connect Function

The connect function is used by a TCP client to establish a connection with a TCP server.

```
#include <sys/socket.h>


int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);


/* Returns: 0 if OK, -1 on error */
```

- *sockfd* is a socket descriptor returned by the socket function.
- The *servaddr* and *addrlen* arguments are a pointer to a socket address structure (which contains the IP address and port number of the server) and its size.

The client does not have to call bind before calling connect: the kernel will choose both an ephemeral port and the source IP address if necessary.

In the case of a TCP socket, the connect function initiates TCP's three-way handshake. The function returns only when the connection is established or an error occurs. There are several different error returns possible:

1. If the client TCP receives no response to its SYN segment, ETIMEDOUT is returned.

    - For example, in 4.4BSD, the client sends one SYN when connect is called, sends another SYN 6 seconds later, and sends another SYN 24 seconds later. If no response is received after a total of 75 seconds, the error is returned.
    - Some systems provide administrative control over this timeout.
2. If the server's response to the client's SYN is a reset (RST), this indicates that no process is waiting for connections on the server host at the port specified (the server process is probably not running). This is a **hard error** and the error ECONNREFUSED is returned to the client as soon as the RST is received. An RST is a type of TCP segment that is sent by TCP when something is wrong. Three conditions that generate an RST are:
    - When a SYN arrives for a port that has no listening server.
    - When TCP wants to abort an existing connection.
    - When TCP receives a segment for a connection that does not exist.
3. If the client's SYN elicits an ICMP "destination unreachable" from some intermediate router, this is considered a **soft error**. The client kernel saves the message but keeps sending SYNs with the same time between each SYN as in the first scenario. If no response is received after some fixed amount of time (75 seconds for 4.4BSD), the saved ICMP error is returned to the process as either EHOSTUNREACH or ENETUNREACH. It is also possible that the remote system is not reachable by any route in the local system's forwarding table, or that the connect call returns without waiting at all. Note that network unreachables are considered obsolete, and applications should just treat ENETUNREACH and EHOSTUNREACH as the same error.

Example: nonexistent host on the local subnet *

We run the client daytimetcpcli and specify an IP address that is on the local subnet (192.168.1/24) but the host ID (100) is nonexistent. When the client host sends out ARP requests (asking for that host to respond with its hardware address), it will never receive an ARP reply.

```
solaris % daytimetcpcli 192.168.1.100
connect error: Connection timed out
```

We only get the error after the connect times out. Notice that our err_sys function prints the human-readable string associated with the ETIMEDOUT error.

Example: no server process running *

We specify a host (a local router) that is not running a daytime server:

```
solaris % daytimetcpcli 192.168.1.5
connect error: Connection refused
```

The server responds immediately with an RST.

Example: destination not reachable on the Internet *

Our final example specifies an IP address that is not reachable on the Internet. If we watch the packets with tcpdump, we see that a router six hops away returns an ICMP host unreachable error.

```
solaris % daytimetcpcli 192.3.4.5
```

```
connect error: No route to host
```

As with the ETIMEDOUT error, connect returns the EHOSTUNREACH error only after waiting its specified amount of time.

In terms of the TCP state transition diagram (Figure 2.4):

- connect moves from the CLOSED state (the state in which a socket begins when it is created by the socket function) to the SYN_SENT state, and then, on success, to the ESTABLISHED state.
- If connect fails, the socket is no longer usable and must be closed. We cannot call connect again on the socket.

In Figure 11.10, we will see that when we call connect in a loop, trying each IP address for a given host until one works, each time connect fails, we must close the socket descriptor and call socket again.

bind Function

The bind function assigns a local protocol address to a socket. The protocol address is the combination of either a 32-bit IPv4 address or a 128-bit IPv6 address, along with a 16-bit TCP or UDP port number.

```
#include <sys/socket.h>


int bind (int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);


/* Returns: 0 if OK,-1 on error */
```

- The second argument *myaddr* is a pointer to a protocol-specific addres
- The third argument *addrlen* is the size of this address structure.

With TCP, calling bind lets us specify a port number, an IP address, both, or neither.

- **Servers bind their well-known port when they start.** (Figure 1.9) If a TCP client or server does not do this, the kernel chooses an ephemeral port for the socket when either connect or listen is called.
    - It is normal for a TCP client to let the kernel choose an ephemeral port, unless the application requires a reserved port (Figure 2.10)
    - However, it is rare for a TCP server to let the kernel choose an ephemeral port, since servers are known by their well-known port.

  Exceptions to this rule are Remote Procedure Call (RPC) servers. They normally let the kernel choose an ephemeral port for their listening socket since this port is then registered with the RPC port mapper. Clients have to contact the port mapper to obtain the ephemeral port before they can connect to the server. This also applies to RPC servers using UDP.

- **A process can bind a specific IP address to its socket.** The IP address must belong to an interface on the host.
    - For a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the socket. Normally, a TCP client does not bind an IP address to its socket. The kernel chooses the source IP address when the socket is connected, based on the outgoing interface that is used, which in turn is based on the route required to reach the server
    - For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address. If a TCP server does not bind an IP address to its socket, the kernel uses the destination IP address of the client's SYN as the server's source IP address.

As mentioned, calling bind lets us specify the IP address, the port, both, or neither. The following table summarizes the values to which we set sin_addr and sin_port, or sin6_addr and sin6_port, depending on the desired result.

IP address	Port	Result
Wildcard	0	Kernel chooses IP address and port
Wildcard	nonzero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	nonzero	Process specifies IP address and port

- If we specify a port number of 0, the kernel chooses an ephemeral port when bind is called.
- If we specify a wildcard IP address, the kernel does not choose the local IP address until either the socket is connected (TCP) or a datagram is sent on the socket (UDP).

Wildcard Address and INADDR_ANY *

With IPv4, the *wildcard* address is specified by the constant INADDR_ANY, whose value is normally 0. This tells the kernel to choose the IP address. Figure 1.9 has the assignment:

```
struct sockaddr_in   servaddr;
servaddr.sin_addr.s_addr = htonl (INADDR_ANY);    /* wildcard */
```

While this works with IPv4, where an IP address is a 32-bit value that can be represented as a simple numeric constant (0 in this case), we cannot use this technique with IPv6, since the 128-bit IPv6 address is stored in a structure. In C we cannot represent a constant structure on the right-hand side of an assignment. To solve this problem, we write:

```
struct sockaddr_in6   serv;

serv.sin6_addr = in6addr_any;    /* wildcard */
```

The system allocates and initializes the in6addr_any variable to the constant IN6ADDR_ANY_INIT. The <netinet/in.h> header contains the extern declaration for in6addr_any.

The value of INADDR_ANY (0) is the same in either network or host byte order, so the use of htonl is not really required. But, since all the INADDR_constants defined by the <netinet/in.h> header are defined in host byte order, we should use htonl with any of these constants.

If we tell the kernel to choose an ephemeral port number for our socket (by specifying a 0 for port number), bind does not return the chosen value. It cannot return this value since the second argument to bind has the const qualifier. To obtain the value of the ephemeral port assigned by the kernel, we must call getsockname to return the protocol address.

Binding a non-wildcard IP address *

A common example of a process binding a non-wildcard IP address to a socket is a host that provides Web servers to multiple organizations:

- First, each organization has its own domain name, such as www.organization.com.
- Next, each organization's domain name maps into a different IP address, but typically on the same subnet.

For example, if the subnet is 198.69.10, the first organization's IP address could be 198.69.10.128, the next 198.69.10.129, and so on. All these IP addresses are then *aliased* onto a single network interface (using the alias option of the ifconfig command on 4.4BSD, for example) so that the IP layer will accept incoming datagrams destined for any of the aliased addresses. Finally, one copy of the HTTP server is started for each organization and each copy binds only the IP address for that organization.

An alternative technique is to run a single server that binds the wildcard address. When a connection arrives, the server calls getsockname to obtain the destination IP address from the client, which in our discussion above could be 198.69.10.128, 198.69.10.129, and so on. The server then handles the client request based on the IP address to which the connection was issued.

One advantage in binding a non-wildcard IP address is that the demultiplexing of a given destination IP address to a given server process is then done by the kernel.

We must be careful to distinguish between the interface on which a packet arrives versus the destination IP address of that packet. In Section 8.8, we will talk about the **weak end system model** and the **strong end system model**. Most implementations employ the former, meaning it is okay for a packet to arrive with a destination IP address that identifies an interface other than the interface on which the packet arrives. (This assumes a multihomed host.) Binding a non-wildcard IP address restricts the datagrams that will be delivered to the socket based only on the destination IP address. It says nothing about the arriving interface, unless the host employs the strong end system model.

A common error from bind is EADDRINUSE ("Address already in use"), which is detailed in Section 7.5 when discussing the SO_REUSEADDR and SO_REUSEPORT socket options.

listen Function

The listen function is called only by a TCP server and it performs two actions:

1. The listen function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. In terms of the TCP state transition diagram (Figure 2.4), the call to listen moves the socket from the CLOSED state to the LISTEN state.
    - When a socket is created by the socket function (and before calling listen), it is assumed to be an active socket, that is, a client socket that will issue a connect.
2. The second argument *backlog* to this function specifies the maximum number of connections the kernel should queue for this socket.

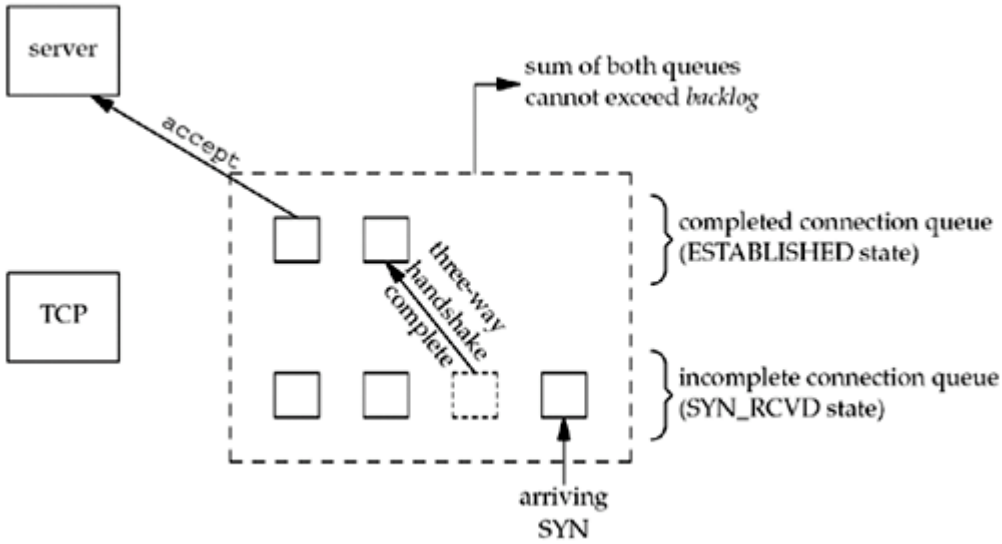
This function is normally called after both the socket and bind functions and must be called before calling the accept function.

Connection queues *

To understand the *backlog* argument, we must realize that for a given listening socket, the kernel maintains two queues:

1. An **incomplete connection queue**, which contains an entry for each SYN that has arrived from a client for which the server is awaiting completion of the TCP three-way handshake. These sockets are in the SYN_RCVD state (Figure 2.4).
2. A **completed connection queue**, which contains an entry for each client with whom the TCP three-way handshake has completed. These sockets are in the ESTABLISHED state (Figure 2.4).

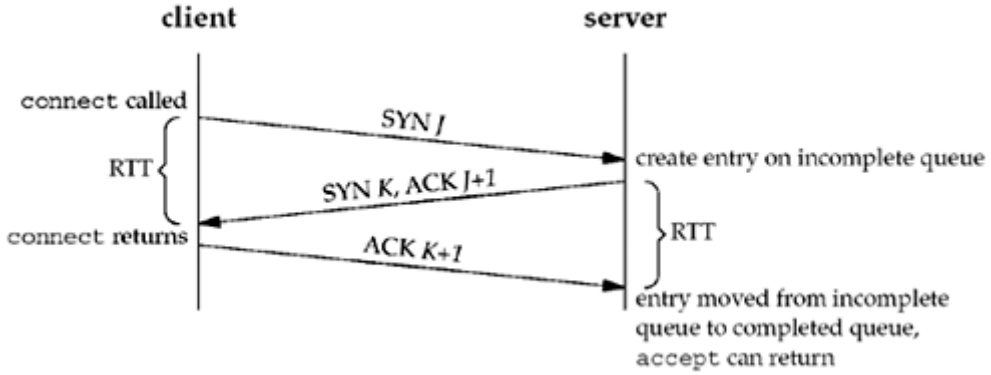
These two queues are depicted in the figure below:



When an entry is created on the incomplete queue, the parameters from the listen socket are copied over to the newly created connection. The connection creation mechanism is completely automatic; the server process is not involved.

Packet exchanges during conenction establishment *

The following figure depicts the packets exchanged during the connection establishment with these two queues:



- When a SYN arrives from a client, TCP creates a new entry on the incomplete queue and then responds with the second segment of the three-way handshake: the server's SYN with an ACK of the client's SYN (Section 2.6).
- This entry will remain on the incomplete queue, until:
    - The third segment of the three-way handshake arrives (the client's ACK of the server's SYN), or
    - The entry times out. (Berkeley-derived implementations have a timeout of 75 seconds for these incomplete entries.)
- If the three-way handshake completes normally, the entry moves from the incomplete queue to the end of the completed queue.
- When the process calls accept:
    - The first entry on the completed queue is returned to the process, or
    - If the queue is empty, the process is put to sleep until an entry is placed onto the completed queue.

The *backlog* argument *

Several points to consider when handling the two queues:

- **Sum of both queues**. The *backlog* argument to the listen function has historically specified the maximum value for the sum of both queues.
- **Multiplied by 1.5**. Berkeley-derived implementations add a fudge factor to the *backlog*: It is multiplied by 1.5.
    - If the *backlog* specifies the maximum number of completed connections the kernel will queue for a socket, then the reason for the fudge factor is to take into account incomplete connections on the queue.
- **Do not specify value of 0** for *backlog*, as different implementations interpret this differently (Figure 4.10). If you do not want any clients connecting to your listening socket, close the listening socket.
- **One RTT**. If the three-way handshake completes normally (no lost segments and no retransmissions), an entry remains on the incomplete connection queue for one RTT.
- **Configurable maximum value**. Many current systems allow the administrator to modify the maximum value for the *backlog*. Historically, sample code always shows a *backlog* of 5 (which is adequate today).

- **What value should the application specify for the *backlog*** (5 is often inadequate)? There is no easy answer to this.
    - HTTP servers now specify a larger value, but if the value specified is a constant in the source code, to increase the constant requires recompiling the server.
    - Another method is to allow a command-line option or an environment variable to override the default. It is always acceptable to specify a value that is larger than supported by the kernel, as the kernel should silently truncate the value to the maximum value that it supports, without returning an error. The following example is the wrapper function for listen which allows the environment variable LISTENQ to override the value specified by the caller:

lib/wrapsock.c#L166

```
void
Listen(int fd, int backlog)
{
   char    *ptr;

      /* can override 2nd argument with environment variable */
   if ( (ptr = getenv("LISTENQ")) != NULL)
      backlog = atoi(ptr);

   if (listen(fd, backlog) < 0)
      err_sys("listen error");
}
/* end Listen */
```

- **Fixed number of connections**. Historically the reason for queuing a fixed number of connections is to handle the case of the server process being busy between successive calls to accept. This implies that of the two queues, the completed queue should normally have more entries than the incomplete queue. Again, busy Web servers have shown that this is false. The reason for specifying a large *backlog* is because the incomplete connection queue can grow as client SYNs arrive, waiting for completion of the three-way handshake.
- **No RST sent if queues are full**. If the queues are full when a client SYN arrives, TCP ignores the arriving SYN; it does not send an RST. This is because the condition is considered temporary, and the client TCP will retransmit its SYN, hopefully finding room on the queue in the near future. If the server TCP immediately responded with an RST, the client's connect would return an error, forcing the application to handle this condition instead of letting TCP's normal retransmission take over. Also, the client could not differentiate between an RST in response to a SYN meaning "there is no server at this port" versus "there is a server at this port but its queues are full."

- **Data queued in the socket's receive buffer**. Data that arrives after the three-way handshake completes, but before the server calls accept, should be queued by the server TCP, up to the size of the connected socket's receive buffer.

The following figure shows actual number of queued connections for values of *backlog*:

backlog	Maximum actual number of queued connections				
	MacOS 10.2.6 AIX 5.1	Linux 2.4.7	HP-UX 11.11	FreeBSD 4.8 FreeBSD 5.1	Solaris 2.9
0	1	3	1	1	1
1	2	4	1	2	2
2	4	5	3	3	4
3	5	6	4	4	5
4	7	7	6	5	6
5	8	8	7	6	8
6	10	9	9	7	10
7	11	10	10	8	11
8	13	11	12	9	13
9	14	12	13	10	14
10	16	13	15	11	16
11	17	14	16	12	17
12	19	15	18	13	19
13	20	16	19	14	20
14	22	17	21	15	22

SYN Flooding *

SYN flooding is a type of attack (the attacker writes a program to send SYNs at a high rate to the victim) that attempts to fill the incomplete connection queue for one or more TCP ports. Additionally, the source IP address of each SYN is set to a random number (called **IP spoofing**) so that the server's SYN/ACK goes nowhere.This also prevents the server from knowing the real IP address of the attacker. By filling the incomplete queue with bogus SYNs, legitimate SYNs are not queued, providing a denial of service to legitimate clients.

The listen's *backlog* argument should specify the maximum number of completed connections for a given socket that the kernel will queue. The purpose ofto limit completed connections is to stop the kernel from accepting new connection requests for a given socket when the application is not accepting them. If a system implements this interpretation, then the application need not specify huge *backlog* values just because the server handles lots of client requests or to provide protection against SYN flooding. The kernel handles lots of incomplete connections, regardless of whether they are legitimate or from a hacker. But even with this interpretation, scenarios do occur where the traditional value of 5 is inadequate.

accept Function

accept is called by a TCP server to return the next completed connection from the front of the completed connection queue (Figure 4.7). If the completed connection queue is empty, the process is put to sleep (assuming the default of a blocking socket).

```
#include <sys/socket.h>


int accept (int sockfd, struct sockaddr *cliaddr, socklen_t *addrlen);
```

```
/* Returns: non-negative descriptor if OK, -1 on error */
```

The *cliaddr* and *addrlen* arguments are used to return the protocol address of the connected peer process (the client). *addrlen* is a value-result argument:

- Before the call, we set the integer value referenced by **addrlen* to the size of the socket address structure pointed to by *cliaddr*;
- On return, this integer value contains the actual number of bytes stored by the kernel in the socket address structure.

If successful, accept returns a new descriptor automatically created by the kernel. This new descriptor refers to the TCP connection with the client.

- The **listening socket** is the first argument (*sockfd*) to accept (the descriptor created by socket and used as the first argument to both bind and listen).
- The **connected socket** is the return value from accept the connected socket.

It is important to differentiate between these two sockets:

- A given server normally creates only one listening socket, which then exists for the lifetime of the server.
- The kernel creates one connected socket for each client connection that is accepted (for which the TCP three-way handshake completes).
- When the server is finished serving a given client, the connected socket is closed.

This function returns up to three values:

- An integer return code that is either a new socket descriptor or an error indication,
- The protocol address of the client process (through the *cliaddr* pointer),
- The size of this address (through the *addrlen* pointer).

If we are not interested in having the protocol address of the client returned, we set both *cliaddr* and *addrlen* to null pointers. See intro/daytimetcpsrv.c.

Example: Value-Result Arguments

The following code shows how to handle the value-result argument to accept by modifying the code from intro/daytimetcpsrv.c to print the IP address and port of the client:

```
#include    "unp.h"
#include    <time.h>

int
main(int argc, char **argv)
{
```

```
    int                listenfd, connfd;
    socklen_t          len;
    struct sockaddr_in  servaddr, cliaddr;
    char               buff[MAXLINE];
    time_t             ticks;

    listenfd = Socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family      = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port        = htons(13);   /* daytime server */

    Bind(listenfd, (SA *) &servaddr, sizeof(servaddr));

    Listen(listenfd, LISTENQ);

    for ( ; ; ) {
        len = sizeof(cliaddr);
        connfd = Accept(listenfd, (SA *) &cliaddr, &len);
        printf("connection from %s, port %d\n",
               Inet_ntop(AF_INET, &cliaddr.sin_addr, buff, sizeof(buff)),
               ntohs(cliaddr.sin_port));

        ticks = time(NULL);
        snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
        Write(connfd, buff, strlen(buff));

        Close(connfd);
    }
}
```

This program does the following:

- **New declarations**. Two new variables are defined:

    - len will be a value-result variable.
    - cliaddr will contain the client's protocol address.
- **Accept connection and print client's address**.
    - Initialize len to the size of the socket address structure
    - Pass a pointer to the cliaddr structure and a pointer to len as the second and third arguments to accept.
    - Call inet_ntop (Section 3.7) to convert the 32-bit IP address in the socket address structure to a dotted-decimal ASCII string and call ntohs (Section 3.4) to convert the 16-bit port number from network byte order to host byte order.

Run this new server and then run our client on the same host, connecting to our server twice in a row:

```
solaris % daytimetcpcli 127.0.0.1
Thu Sep 11 12:44:00 2003
solaris % daytimetcpcli 192.168.1.20
Thu Sep 11 12:44:09 2003
```

We first specify the server's IP address as the loopback address (127.0.0.1) and then as its own IP address (192.168.1.20). Here is the corresponding server output:

```
solaris # daytimetcpsrv1
connection from 127.0.0.1, port 43388
connection from 192.168.1.20, port 43389
```

Since our daytime client does not call bind, the kernel chooses the source IP address based on the outgoing interface that is used.

- In the first case, the kernel sets the source IP address to the loopback address;
- In the second case, it sets the address to the IP address of the Ethernet interface.

We can also see in this example that the ephemeral port chosen by the Solaris kernel is 43388, and then 43389

The pound sign (#) as the shell prompt indicates that our server must run with superuser privileges to bind the reserved port of 13. If we do not have superuser privileges, the call to bind will fail:

```
solaris % daytimetcpsrv1
bind error: Permission denied
```

fork and exec Functions

Concurrent Servers

The server described in intro/daytimetcpsrv1.c is an **iterative server**. But when a client request can take longer to service, we do not want to tie up a single server with one client; we want to handle multiple clients at the same time. The simplest way to write a concurrent server under Unix is to fork a child process to handle each client.

The following code shows the outline for a typical concurrent server:

```
pid_t pid;
int   listenfd,  connfd;

listenfd = Socket( ... );

   /* fill in sockaddr_in{} with server's well-known port */
Bind(listenfd, ... );
Listen(listenfd, LISTENQ);

for ( ; ; ) {
   connfd = Accept (listenfd, ... );    /* probably blocks */

   if( (pid = Fork()) == 0) {
      Close(listenfd);    /* child closes listening socket */
      doit(connfd);      /* process the request */
      Close(connfd);     /* done with this client */
      exit(0);          /* child terminates */
   }

   Close(connfd);       /* parent closes connected socket */
}
```

- When a connection is established, accept returns, the server calls fork, and the child process services the client (on connfd, the connected socket) and the parent process waits for another connection (on listenfd, the listening socket). The parent closes the connected socket since the child handles the new client.
- We assume that the function doit does whatever is required to service the client. When this function returns, we explicitly close the connected socket in the child. This is not required since the next statement calls exit, and part of process termination is to close all open descriptors by the kernel. Whether to include this explicit call to close or not is a matter of personal programming taste.

Reference count of sockets *

Calling close on a TCP socket causes a FIN to be sent, followed by the normal TCP connection termination sequence. Why doesn't the close of connfd by the parent terminate its connection with the client? To understand what's happening, we must understand that every file or socket has a **reference count**. The reference count is maintained in the file table entry. This is a count of the number of descriptors that are currently open that refer to this file or socket. In the above code:

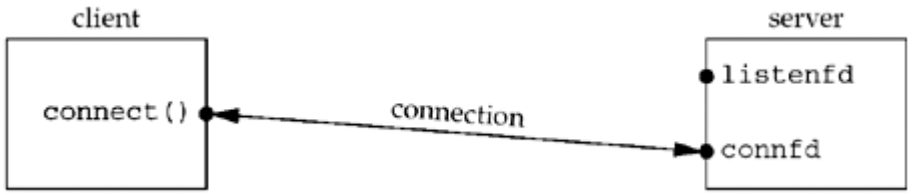
- After socket returns, the file table entry associated with listenfd has a reference count of 1.
- After accept returns, the file table entry associated with connfd has a reference count of 1.
- But, after fork returns, both descriptors are shared (duplicated) between the parent and child, so the file table entries associated with both sockets now have a reference count of 2. Therefore, when the parent closes connfd, it just decrements the reference count from 2 to 1 and that is all.
- The actual cleanup and de-allocation of the socket does not happen until the reference count reaches 0. This will occur at some time later when the child closes connfd.

Visualizing the sockets and connection *

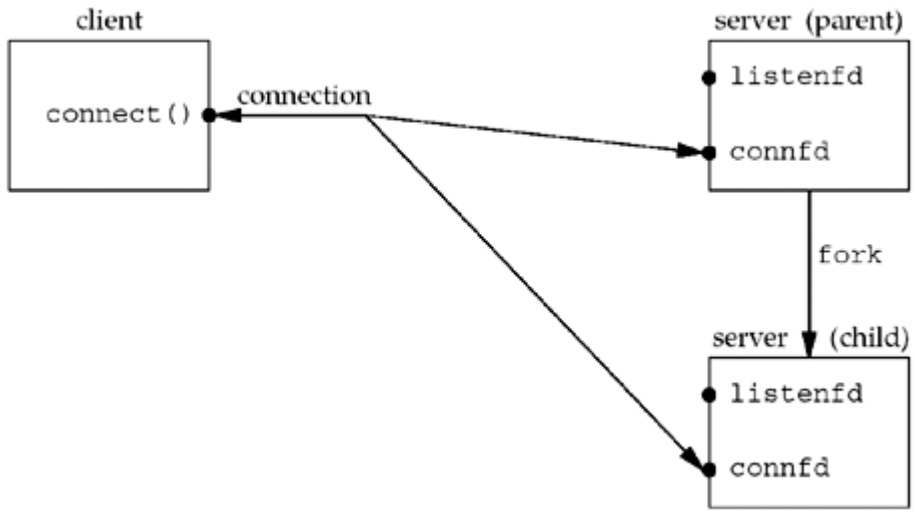
The following figures visualize the sockets and connection in the code above:

Before call to accept returns, the server is blocked in the call to accept and the connection request arrives from the client:

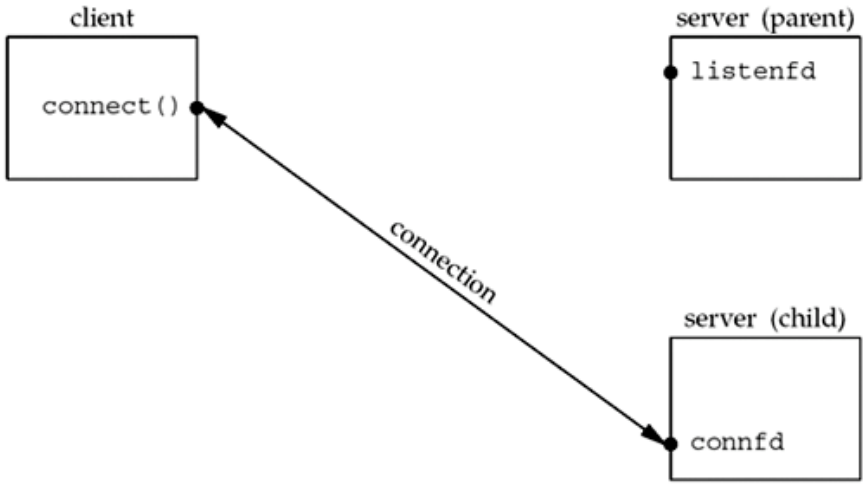
Ater return from accept, the connection is accepted by the kernel and a new socket connfd is created (this is a connected socket and data can now be read and written across the connection):



After fork returns, both descriptors, listenfd and connfd, are shared (duplicated) between the parent and child:



After the parent closes the connected socket and the child closes the listening socket:



This is the desired final state of the sockets. The child is handling the connection with the client and the parent can call accept again on the listening socket, to handle the next client connection.

close Function

The normal Unix close function is also used to close a socket and terminate a TCP connection.

```
#include <unistd.h>

int close (int sockfd);

/* Returns: 0 if OK, -1 on error */
```

The default action of close with a TCP socket is to mark the socket as closed and return to the process immediately. The socket descriptor is no longer usable by the process: It cannot be used as an argument to read or write. But, TCP will try to send any data that is already queued to be sent

to the other end, and after this occurs, the normal TCP connection termination sequence takes place.

SO_LINGER socket option lets us change this default action with a TCP socket.

Descriptor Reference Counts

As mentioned, when the parent process in our concurrent server closes the connected socket, this just decrements the reference count for the descriptor. Since the reference count was still greater than 0, this call to close did not initiate TCP's four-packet connection termination sequence. This is the behavior we want with our concurrent server with the connected socket that is shared between the parent and child.

If we really want to send a FIN on a TCP connection, the shutdown function can be used instead of close.

Be aware if the parent does not call close for each connected socket returned by accept:

1. **The parent will eventually run out of descriptors** (there is usually a limit to the number of descriptors that any process can have open at any time)
2. **None of the client connections will be terminated.** When the child closes the connected socket, its reference count will go from 2 to 1 and it will remain at 1 since the parent never closes the connected socket. This will prevent TCP's connection termination sequence from occurring, and the connection will remain open.

getsockname and getpeername Functions

- getsockname returns the local protocol address associated with a socket.
- getpeername returns the foreign protocol address associated with a socket.

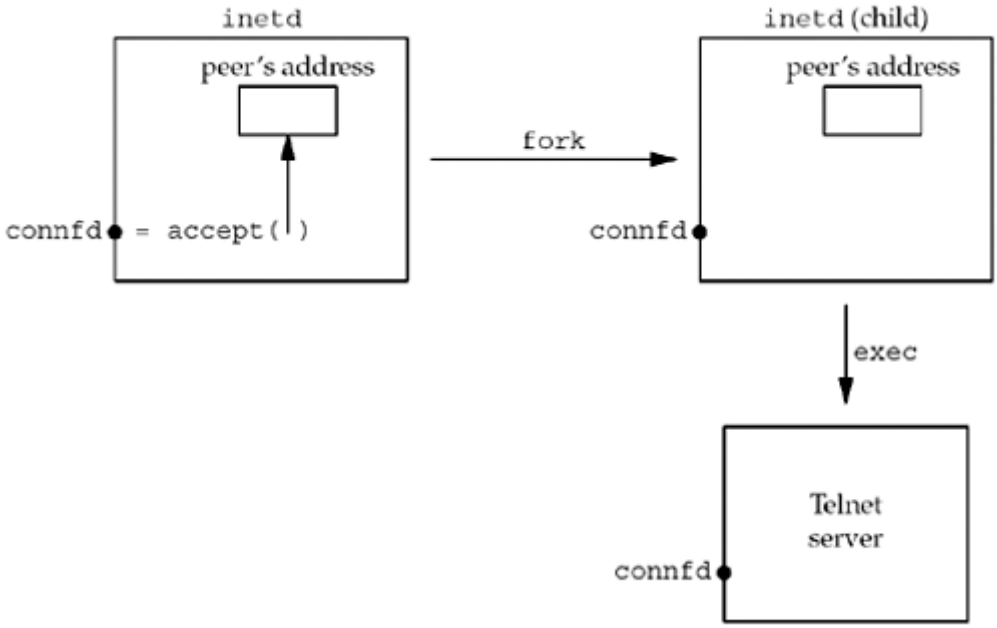
```
#include <sys/socket.h>


int getsockname(int sockfd, struct sockaddr *localaddr, socklen_t *addrlen);

int getpeername(int sockfd, struct sockaddr *peeraddr, socklen_t *addrlen);


/* Both return: 0 if OK, -1 on error */
```

The *addrlen* argument for both functions is value-result argument: both functions fill in the socket address structure pointed to by localaddr or peeraddr.

The term "name" in the function name is misleading. These two functions return the protocol address associated with one of the two ends of a network connection, which for IPV4 and IPV6 is the combination of an IP address and port number. These functions have nothing to do with domain names.

These two functions are required for the following reasons:

- After connect successfully returns in a TCP client that does not call bind, getsockname returns the local IP address and local port number assigned to the connection by the kernel.
- After calling bind with a port number of 0 (telling the kernel to choose the local port number), getsockname returns the local port number that was assigned.
- getsockname can be called to obtain the address family of a socket.
- In a TCP server that binds the wildcard IP address (intro/daytimetcpsrv.c), once a connection is established with a client (accept returns successfully), the server can call getsockname to obtain the local IP address assigned to the connection. The socket descriptor argument to getsockname must be that of the connected socket, and not the listening socket.
- When a server is execed by the process that calls accept, the only way the server can obtain the identity of the client is to call getpeername. For example, inetd forks and execs a TCP server (follwing figure):
    - inetd calls accept, which return two values: the connected socket descriptor (connfd, return value of the function) and the "peer's address" (an Internet socket address structure) that contains the IP address and port number of the client.
    - fork is called and a child of inetd is created, with a copy of the parent's memory image, so the socket address structure is available to the child, as is the connected socket descriptor (since the descriptors are shared between the parent and child).
    - When the child execs the real server (e.g. Telnet server that we show), the memory image of the child is replaced with the new program file for the Telnet server (the socket address structure containing the peer's address is lost), and the connected socket descriptor remains open across the exec. One of the first function calls performed by the Telnet server is getpeername to obtain the IP address and port number of the client.



In this example, the Telnet server must know the value of connfd when it starts. There are two common ways to do this.

1. The process calling exec pass it as a command-line argument to the newly execed program.

2. A convention can be established that a certain descriptor is always set to the connected socket before calling exec.

The second one is what inetd does, always setting descriptors 0, 1, and 2 to be the connected socket.

Example: Obtaining the Address Family of a Socket

The sockfd_to_family function shown in the code below returns the address family of a socket.

lib/sockfd_to_family.c

```
int
sockfd_to_family(int sockfd)
{
   struct sockaddr_storage ss;
   socklen_t   len;

   len = sizeof(ss);
   if (getsockname(sockfd, (SA *) &ss, &len) < 0)
      return(-1);
   return(ss.ss_family);
}
```

This program does the following:

- **Allocate room for largest socket address structure**. Since we do not know what type of socket address structure to allocate, we use a sockaddr_storage value, since it can hold any socket address structure supported by the system.
- **Call getsockname.** We call getsockname and return the address family. The POSIX specification allows a call to getsockname on an unbound socket.

Socket Programming

How do we build Internet applications? Socket programming is the key API for programming distributed applications on the Internet.

Socket program is a key skill needed for the robotics project for exerting control - in this case the controller running on your laptop will connect to the server running on the bot.

Goals

We plan to learn the following:

- What is a socket?
- The client-server model
- Byte order
- TCP socket API
- Concurrent server design
- Example of echo client and iterative server
- Example of echo client and concurrent server

The basics

Program. A program is an executable file residing on a disk in a directory. A program is read into memory and is executed by the kernel as a result of an exec() function. The exec() has six variants, but we only consider the simplest one (exec()) in this course.

Process. An executing instance of a program is called a *process*. Sometimes, *task* is used instead of process with the same meaning. UNIX guarantees that every process has a unique identifier called the *process ID*. The process ID is always a non-negative integer.

File descriptors. File descriptors are normally small non-negative integers that the kernel uses to identify the files being accessed by a particular process. Whenever it opens an existing file or creates a new file, the kernel returns a file descriptor that is used to read or write the file. As we will see in this course, sockets are based on a very similar mechanism (socket descriptors).

The client-server model

The client-server model is one of the most used communication paradigms in networked systems. Clients normally communicates with one server at a time. From a server's perspective, at any point in time, it is not unusual for a server to be communicating with multiple clients. Client need to know of the existence of and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established

Client and servers communicate by means of multiple layers of network protocols. In this course we will focus on the TCP/IP protocol suite.

The scenario of the client and the server on the same local network (usually called LAN, Local Area Network) is shown in Figure 1

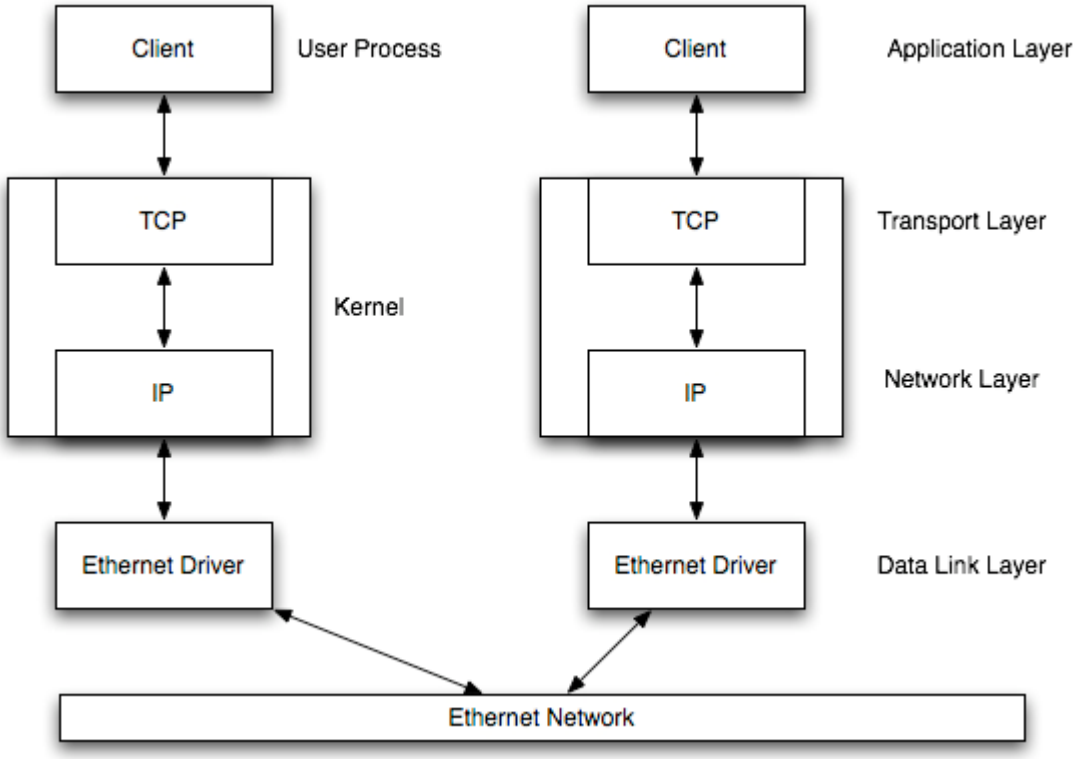


Figure 1: Client and server on the same Ethernet communicating using TCP/IP.

The client and the server may be in different LANs, with both LANs connected to a Wide Area Network (WAN) by means of *routers*. The largest WAN is the Internet, but companies may have their own WANs. This scenario is depicted in Figure 2.

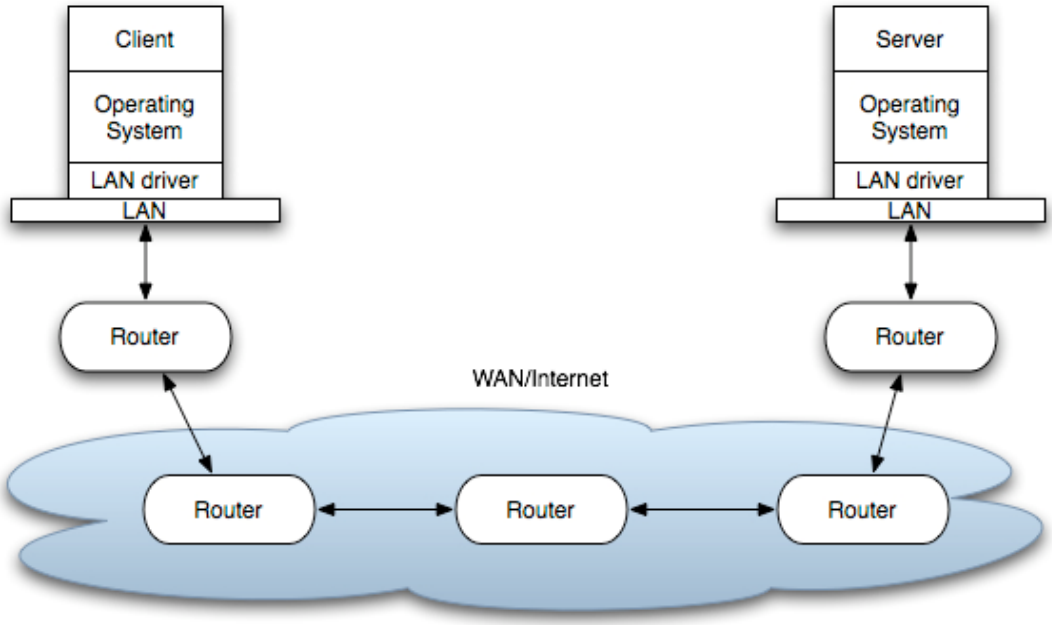


Figure 2: Client and server on different LANs connected through WAN/Internet.

The flow of information between the client and the server goes down the protocol stack on one side, then across the network and then up the protocol stack on the other side.

Transmission Control Protocol (TCP)

TCP provides a *connection oriented service*, since it is based on connections between clients and servers.

TCP provides reliability. When a TCP client send data to the server, it requires an acknowledgement in return. If an acknowledgement is not received, TCP automatically retransmit the data and waits for a longer period of time.

TCP is instead a byte-stream protocol, without any boundaries at all.

TCP is described in RFC 793, RFC 1323, RFC 2581 and RFC 3390.

Socket addresses

IPv4 socket address structure is named sockaddr_in and is defined by including the <netinet/in.h> header.

The POSIX definition is the following:

```
struct in_addr{
in_addr_t s_addr;                    /*32 bit IPv4 network byte ordered address*/
};


struct sockaddr_in {
  uint8_t sin_len; /* length of structure (16)*/
  sa_family_t sin_family; /* AF_INET*/
  in_port_t sin_port; /* 16 bit TCP or UDP port number */
  struct in_addr sin_addr; /* 32 bit IPv4 address*/
  char sin_zero[8]; /* not used but always set to zero */
};
```

The uint8_t datatype is unsigned 8-bit integer.

Generic Socket Address Structure

A socket address structure is always passed by reference as an argument to any socket functions. But any socket function that takes one of these pointers as an argument must deal with socket address structures from any of the supported protocol families.

A problem arises in declaring the type of pointer that is passed. With ANSI C, the solution is to use void * (the generic pointer type). But the socket functions predate the definition of ANSI C and the solution chosen was to define a generic socket address as follows:

```
struct sockaddr {
  uint8_t sa_len;
  sa_family_t sa_family; /* address family: AD_xxx value */
  char sa_data[14];
};
```

Host Byte Order to Network Byte Order Conversion

There are two ways to store two bytes in memory: with the lower-order byte at the starting address (*little-endian* byte order) or with the high-order byte at the starting address (*big-endian* byte order). We call them collectively *host byte order*. For example, an Intel processor stores the 32-bit integer as four consecutives bytes in memory in the order 1-2-3-4, where 1 is the most significant byte. IBM PowerPC processors would store the integer in the byte order 4-3-2-1.

Networking protocols such as TCP are based on a specific *network byte order*. The Internet protocols use big-endian byte ordering.

The htons(), htonl(), ntohs(), and ntohl() Functions

The follwowing functions are used for the conversion:

```
#include <netinet/in.h>

uint16_t htons(uint16_t host16bitvalue);

uint32_t htonl(uint32_t host32bitvalue);

uint16_t ntohs(uint16_t net16bitvalue);

uint32_t ntohl(uint32_t net32bitvalue);
```

The first two return the value in network byte order (16 and 32 bit, respectively). The latter return the value in host byte order (16 and 32 bit, respectively).

TCP Socket API

The sequence of function calls for the client and a server participating in a TCP connection is presented in Figure 3.

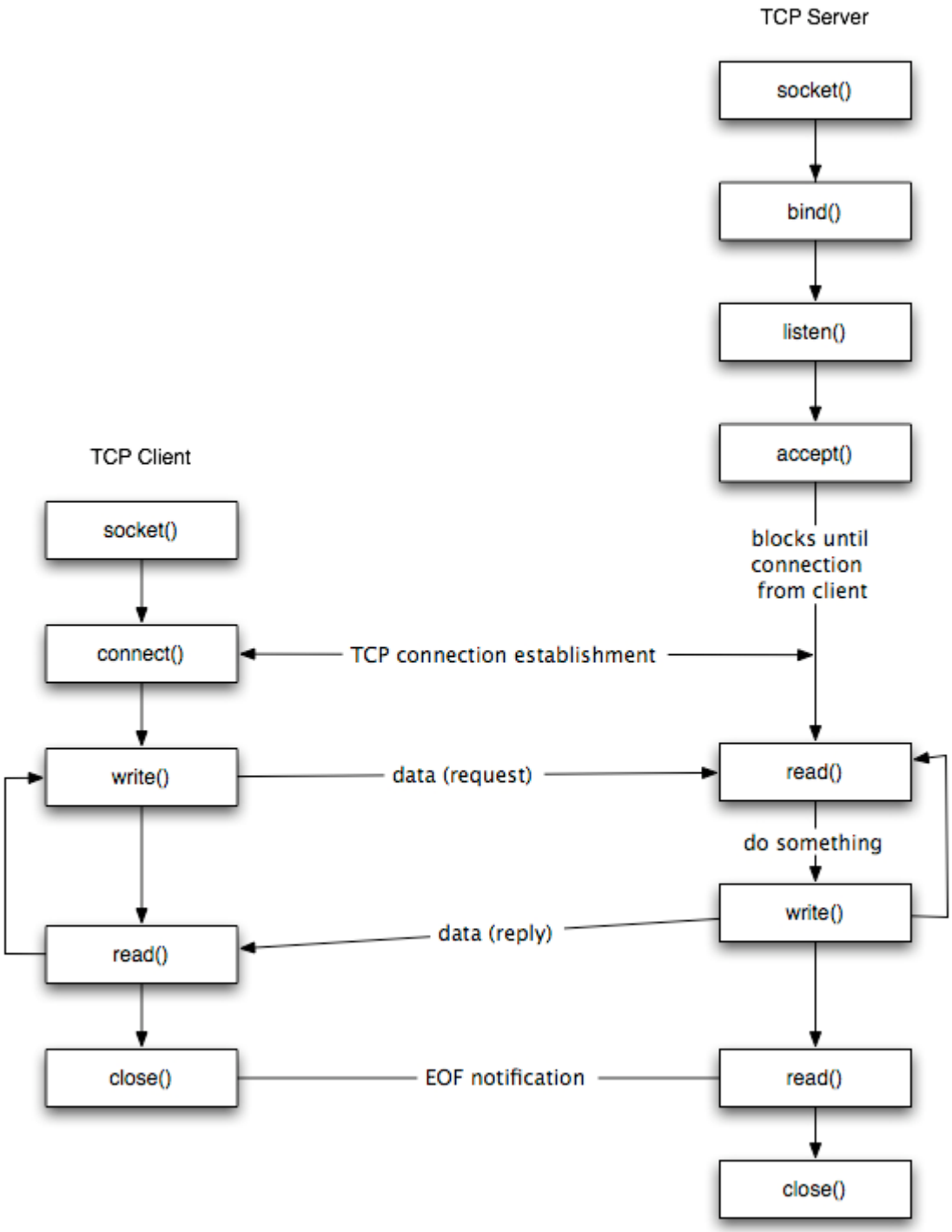


Figure 3: TCP client-server.

As shown in the figure, the steps for establishing a TCP socket on the client side are the following:

- Create a socket using the socket() function;
- Connect the socket to the address of the server using the connect() function;

- Send and receive data by means of the read() and write() functions.

The steps involved in establishing a TCP socket on the server side are as follows:

- Create a socket with the socket() function;
- Bind the socket to an address using the bind() function;
- Listen for connections with the listen() function;
- Accept a connection with the accept() function system call. This call typically blocks until a client connects with the server.
- Send and receive data by means of send() and receive().

The socket() Function

The first step is to call the socket function, specifying the type of communication protocol (TCP based on IPv4, TCP based on IPv6, UDP).

The function is defined as follows:

```
#include <sys/socket.h>

int socket (int family, int type, int protocol);
```

where family specifies the protocol family (AF_INET for the IPv4 protocols), type is a constant described the type of socket (SOCK_STREAM for stream sockets and SOCK_DGRAM for datagram sockets.

The function returns a non-negative integer number, similar to a file descriptor, that we define *socket descriptor* or -1 on error.

The connect() Function

The connect() function is used by a TCP client to establish a connection with a TCP server/

The function is defined as follows:

```
#include <sys/socket.h>

int connect (int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

where sockfd is the socket descriptor returned by the socket function.

The function returns 0 if the it succeeds in establishing a connection (i.e., successful TCP three-way handshake, -1 otherwise.

The client does not have to call bind() in Section before calling this function: the kernel will choose both an ephemeral port and the source IP if necessary.

The bind() Function

The bind() assigns a local protocol address to a socket. With the Internet protocols, the address is the combination of an IPv4 or IPv6 address (32-bit or 128-bit) address along with a 16 bit TCP port number.

The function is defined as follows:

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

where sockfd is the socket descriptor, myaddr is a pointer to a protocol-specific address and addrlen is the size of the address structure.

bind() returns 0 if it succeeds, -1 on error.

This use of the generic socket address sockaddr requires that any calls to these functions must cast the pointer to the protocol-specific address structure. For example for and IPv4 socket structure:

```
struct sockaddr_in serv; /* IPv4 socket address structure */

bind(sockfd, (struct sockaddr*) &serv, sizeof(serv))
```

A process can bind a specific IP address to its socket: for a TCP client, this assigns the source IP address that will be used for IP datagrams sent on the sockets. For a TCP server, this restricts the socket to receive incoming client connections destined only to that IP address.

Normally, a TCP client does not bind an IP address to its socket. The kernel chooses the source IP socket is connected, based on the outgoing interface that is used. If a TCP server does not bind an IP address to its socket, the kernel uses the destination IP address of the incoming packets as the server's source address.

bind() allows to specify the IP address, the port, both or neither.

The table below summarizes the combinations for IPv4.

IP Address	IP Port	Result
INADDR_ANY	0	Kernel chooses IP address and port
INADDR_ANY	non zero	Kernel chooses IP address, process specifies port
Local IP address	0	Process specifies IP address, kernel chooses port
Local IP address	non zero	Process specifies IP address and port

The listen() Function

The listen() function converts an unconnected socket into a passive socket, indicating that the kernel should accept incoming connection requests directed to this socket. It is defined as follows:

```
#include <sys/socket.h>

int listen(int sockfd, int backlog);
```

where sockfd is the socket descriptor and backlog is the maximum number of connections the kernel should queue for this socket. The backlog argument provides an hint to the system of the number of outstanding connect requests that is should enqueue in behalf of the process. Once the queue is full, the system will reject additional connection requests. The backlog value must be chosen based on the expected load of the server.

The function listen() return 0 if it succeeds, -1 on error.

The accept() Function

The accept() is used to retrieve a connect request and convert that into a request. It is defined as follows:

```
#include <sys/socket.h>

int accept(int sockfd, struct sockaddr *cliaddr,
socklen_t *addrlen);
```

where sockfd is a new file descriptor that is connected to the client that called the connect(). The cliaddr and addrlen arguments are used to return the protocol address of the client. The new socket descriptor has the same socket type and address family of the original socket. The original socket passed to accept() is not associated with the connection, but instead remains available to receive additional

connect requests. The kernel creates one connected socket for each client connection that is accepted.

If we don't care about the client's identity, we can set the cliaddr and addrlen to NULL. Otherwise, before calling the accept function, the cliaddr parameter has to be set to a buffer large enough to hold the address and set the interger pointed by addrlen to the size of the buffer.

The send() Function

Since a socket endpoint is represented as a file descriptor, we can use read and write to communicate with a socket as long as it is connected. However, if we want to specify options we need another set of functions.

For example, send() is similar to write() but allows to specify some options. send() is defined as follows:

```
#include <sys/socket.h>
ssize_t send(int sockfd, const void *buf, size_t nbytes, int flags);
```

where buf and nbytes have the same meaning as they have with write. The additional argument flags is used to specify how we want the data to be transmitted. We will not consider the possible options in this course. We will assume it equal to 0.

The function returns the number of bytes if it succeeds, -1 on error.

The receive() Function

The recv() function is similar to read(), but allows to specify some options to control how the data are received. We will not consider the possible options in this course. We will assume it is equal to 0.

receive is defined as follows:

```
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buf, size_t nbytes, int flags);
```

The function returns the length of the message in bytes, 0 if no messages are available and peer had done an orderly shutdown, or -1 on error.

The close() Function

The normal close() function is used to close a socket and terminate a TCP socket. It returns 0 if it succeeds, -1 on error. It is defined as follows:

```
#include <unistd.h>

int close(int sockfd);
```

Concurrent Servers

There are two main classes of servers, iterative and concurrent. An *iterative* server iterates through each client, handling it one at a time. A *concurrent* server handles multiple clients at the same time. The simplest technique for a concurrent server is to call the fork function, creating one child process for each client. An alternative technique is to use *threads* instead (i.e., light-weight processes).

The fork() function

The fork() function is the only way in Unix to create a new process. It is defined as follows:

```
#include <unist.h>

pid_t fork(void);
```

The function returns 0 if in child and the process ID of the child in parent; otherwise, -1 on error.

In fact, the function fork() is called once but returns *twice*. It returns once in the calling process (called the parent) with the process ID of the newly created process (its child). It also returns in the child, with a return value of 0. The return value tells whether the current process is the parent or the child.

Example

A typical concurrent server has the following structure:

```
pid_t pid;
int listenfd, connfd;
listenfd = socket(...);
```

```
/***fill the socket address with server's well known port***/

bind(listenfd, ...);
listen(listenfd, ...);

for ( ; ; ) {

  connfd = accept(listenfd, ...); /* blocking call */

  if ( (pid = fork()) == 0 ) {

    close(listenfd); /* child closes listening socket */

    /***process the request doing something using connfd ***/
    /* ..... */

    close(connfd);
    exit(0);  /* child terminates
   }
   close(connfd);  /*parent closes connected socket*/
}
}
```

When a connection is established, accept returns, the server calls fork, and the child process services the client (on the connected socket connfd). The parent process waits for another connection (on the listening socket listenfd. The parent closes the connected socket since the child handles the new client. The interactions among client and server are presented in Figure 4.

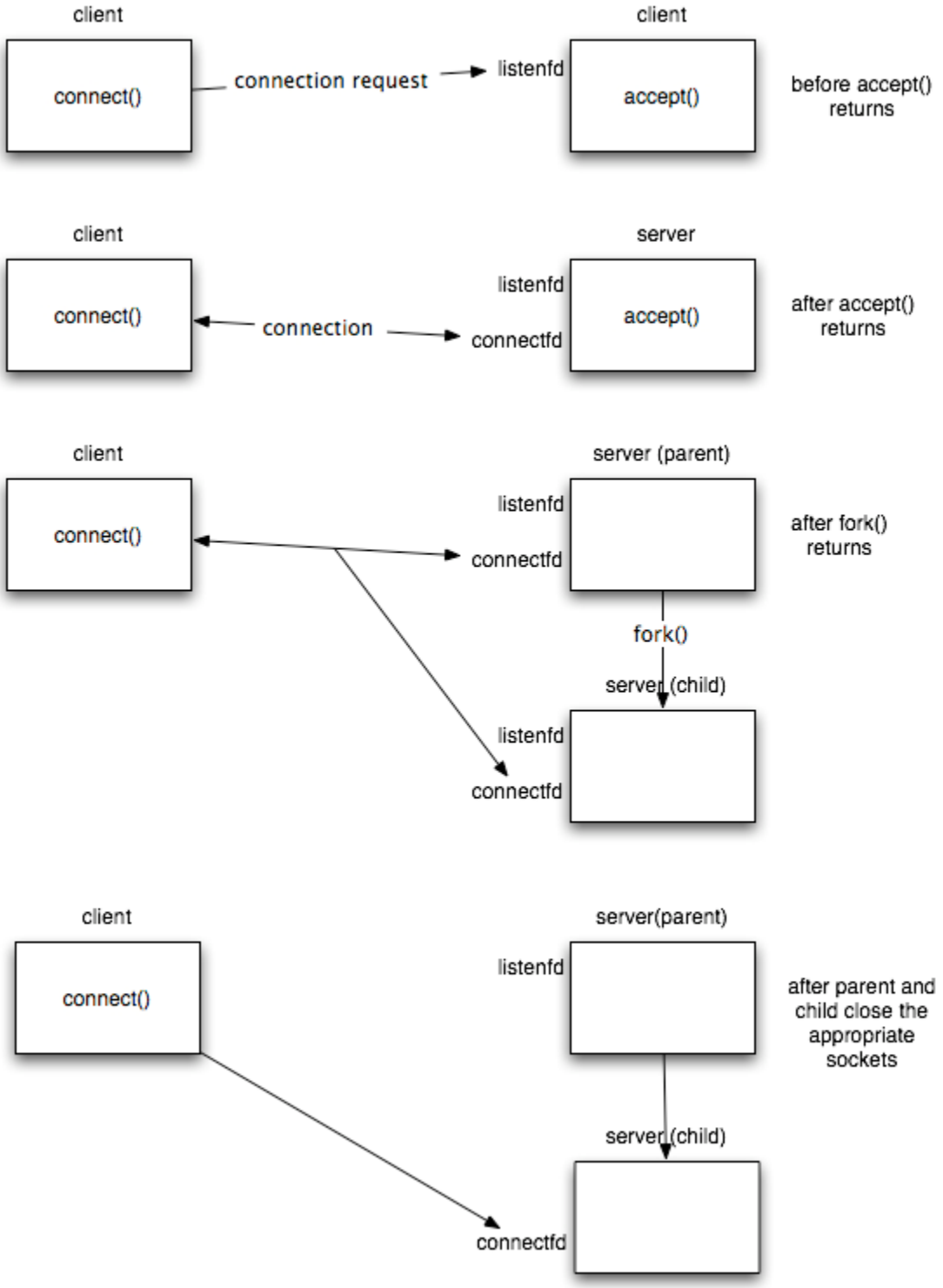


Figure 4: Example of interaction among a client and a concurrent server.

TCP Client/Server Examples

We now present a complete example of the implementation of a TCP based echo server to summarize the concepts presented above. We present an iterative and a concurrent implementation of the server.

We recommend that you run the client and server on different machines so there is a TCP connection over the Internet. However, you can also use a local TCP connection bewteen the client and server processes using the IP address **127.0.0.1** as the address given to the client. The **localhost** (meaning ”this computer”) is the standard hostname given to the address of the loopback network interface.

Please note that socket programming regularly resolve names of machines such as wildcat.cs.dartmouth.edu to a 32 bit IP address needed to make a connect(). In class we have interacted directly with the DNS (domain name server) using the host command:

```
$# you can use localhost or 127.0.0.1 for testing the client and server on the same
 machine

$ host localhost
localhost has address 127.0.0.1

$# find the name of the machine you are logged into

$ hostname
bear.cs.dartmouth.edu

$# find the IP address of the machine

$ host bear
bear.cs.dartmouth.edu has address 129.170.213.32
bear.cs.dartmouth.edu mail is handled by 0 mail.cs.dartmouth.edu.

$# If you have the dot IP address form you can find the name

$ host 129.170.213.32
32.213.170.129.in-addr.arpa domain name pointer bear.cs.dartmouth.edu.
```

Host allows us to get the host IP address by name or get the host name given the IP address.

Luckly you don’t have to call “host” from your code. There are two commands that you can use:

```
struct hostent *gethostbyname(const char *name);

struct hostent *gethostbyaddr(const char *addr,  int len, int type);
```

TCP Echo Client

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

#define MAXLINE 4096 /*max text line length*/
#define SERV_PORT 3000 /*port*/

int
main(int argc, char **argv)
{
 int sockfd;
 struct sockaddr_in servaddr;
 char sendline[MAXLINE], recvline[MAXLINE];

 //basic check of the arguments
 //additional checks can be inserted
 if (argc !=2) {
  perror("Usage: TCPClient <IP address of the server");
  exit(1);
 }

 //Create a socket for the client
 //If sockfd<0 there was an error in the creation of the socket
 if ((sockfd = socket (AF_INET, SOCK_STREAM, 0)) <0) {
  perror("Problem in creating the socket");
  exit(2);
```

```
}

//Creation of the socket
memset(&servaddr, 0, sizeof(servaddr));
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr= inet_addr(argv[1]);
servaddr.sin_port =  htons(SERV_PORT); //convert to big-endian order

//Connection of the client to the socket
if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr))<0) {
 perror("Problem in connecting to the server");
 exit(3);
}

while (fgets(sendline, MAXLINE, stdin) != NULL) {

 send(sockfd, sendline, strlen(sendline), 0);

 if (recv(sockfd, recvline, MAXLINE,0) == 0){
  //error: server terminated prematurely
  perror("The server terminated prematurely");
  exit(4);
 }
 printf("%s", "String received from the server: ");
 fputs(recvline, stdout);
}

 exit(0);
}
```

TCP Iterative Server

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

```
#define MAXLINE 4096 /*max text line length*/
#define SERV_PORT 3000 /*port*/
#define LISTENQ 8 /*maximum number of client connections */

int main (int argc, char **argv)
{
 int listenfd, connfd, n;
 socklen_t clilen;
 char buf[MAXLINE];
 struct sockaddr_in cliaddr, servaddr;

 //creation of the socket
 listenfd = socket (AF_INET, SOCK_STREAM, 0);

 //preparation of the socket address
 servaddr.sin_family = AF_INET;
 servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
 servaddr.sin_port = htons(SERV_PORT);

 bind (listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

 listen (listenfd, LISTENQ);

 printf("%s\n","Server running...waiting for connections.");

 for ( ; ; ) {

  clilen = sizeof(cliaddr);
  connfd = accept (listenfd, (struct sockaddr *) &cliaddr, &clilen);
  printf("%s\n","Received request...");

  while ( (n = recv(connfd, buf, MAXLINE,0)) > 0)  {
   printf("%s","String received from and resent to the client:");
   puts(buf);
   send(connfd, buf, n, 0);
  }

 if (n < 0) {
  perror("Read error");
  exit(1);
 }
 close(connfd);
```

```
 }
 //close listening socket
 close (listenfd);
}
```

Localhost Execution of Client/Server

To run the client and server try the following. It is best if you can run the server and client on different machines. But we will first show how to test the client and server on the same host using the locahost 127.0.0.1

```
$# first mygcc the client and server

$ mygcc -o echoClient echoClient.c
$ mygcc -o echoServer echoServer.c

$# first run the server in background

$ ./echoServer&
[1] 341
$ Server running...waiting for connections.

$ #Now connect using the localhost address 127.0.0.1 and then type something
$ # the control C out of the client and ps and kill the server

$ ./echoClient 127.0.0.1
Received request...
Hello CS23!
String received from and resent to the client:Hello CS23!

String received from the server: Hello CS23!
^C
$ ps
  PID TTY          TIME CMD
  208 ttys000    0:00.04 -bash
  341 ttys000    0:00.00 ./echoServer
  236 ttys001    0:00.01 -bash
$ kill -9 341
$
[1]+  Killed                 ./echoServer
```

Remote Execution of Client/Server

Now lets do the same thing but run the server on a remote machine and client locally. This time we will have to use the **host** command to find the IP address of the host we run the server on. The rest is the same as the localhost example above.

First, we ssh into bear and run the server and get the local IP address of bear

```
$ssh campbell@bear.cs.dartmouth.edu
campbell@bear.cs.dartmouth.edu's password:
Last login: Sun Feb 14 23:27:30 2010 from c-71-235-190-26.hsd1.ct.comcast.net
$ cd public_html/cs23
$ mygcc -o echoServer echoServer.c
$ ./echoServer&
[1] 6020
$ Server running...waiting for connections.

$ host bear
bear.cs.dartmouth.edu has address 129.170.213.32
bear.cs.dartmouth.edu mail is handled by 0 mail.cs.dartmouth.edu.
```

Next, we start the client on our local machine and type something. We terminate the same way as before

First, we ssh into bear and run the server and get the local IP address of bear

```
$# Just to show we are running on a different machine

$ hostname
andrew-campbells-macbook-pro.local

$ ./echoClient 129.170.213.32
Hello CS23!
String received from the server: Hello CS23!
^C
```

Notice, that when we type make a connection and type in "Hello CS23!" we get the following at the server.

```
$# Just to show we are running on a different machine
```

```
$ Received request...
String received from and resent to the client:Hello CS23!

$# Now we clean up

$ ps
  PID TTY          TIME CMD
 5972 pts/2    00:00:00 bash
 6020 pts/2    00:00:00 echoServer
 6040 pts/2    00:00:00 ps
$ kill -9 6020
$
[1]+  Killed                 ./echoServer
```

TCP Concurrent Echo Server

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

#define MAXLINE 4096 /*max text line length*/
#define SERV_PORT 3000 /*port*/
#define LISTENQ 8 /*maximum number of client connections*/

int main (int argc, char **argv)
{
 int listenfd, connfd, n;
 pid_t childpid;
 socklen_t clilen;
 char buf[MAXLINE];
 struct sockaddr_in cliaddr, servaddr;

 //Create a socket for the soclet
 //If sockfd<0 there was an error in the creation of the socket
```

```
if ((listenfd = socket (AF_INET, SOCK_STREAM, 0)) <0) {
 perror("Problem in creating the socket");
 exit(2);
}


//preparation of the socket address
servaddr.sin_family = AF_INET;
servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
servaddr.sin_port = htons(SERV_PORT);

//bind the socket
bind (listenfd, (struct sockaddr *) &servaddr, sizeof(servaddr));

//listen to the socket by creating a connection queue, then wait for clients
listen (listenfd, LISTENQ);

printf("%s\n","Server running...waiting for connections.");

for ( ; ; ) {

 clilen = sizeof(cliaddr);
 //accept a connection
 connfd = accept (listenfd, (struct sockaddr *) &cliaddr, &clilen);

 printf("%s\n","Received request...");

 if ( (childpid = fork ()) == 0 ) {//if it's 0, it's child process

 printf ("%s\n","Child created for dealing with client requests");

 //close listening socket
 close (listenfd);

 while ( (n = recv(connfd, buf, MAXLINE,0)) > 0)  {
  printf("%s","String received from and resent to the client:");
  puts(buf);
  send(connfd, buf, n, 0);
 }

 if (n < 0)
  printf("%s\n", "Read error");
```

```
 exit(0);
 }
 //close socket of the server
 close(connfd);
}
}
```

Remote Execution of concurrent Client/Server

Now, we run the server on a remote machine and then run two clients talking to the same server. We use hostname so we know what machines we use in the example below.

First, we start the concurrent server on a remote machine and get its IP address that the clients will use.

```
$ mygcc -o conEchoServer conEchoServer.c
$ ./conEchoServer&
[1] 6075
$ Server running...waiting for connections.

$ hostname
bear.cs.dartmouth.edu
$ host bear
bear.cs.dartmouth.edu has address 129.170.213.32
bear.cs.dartmouth.edu mail is handled by 0 mail.cs.dartmouth.edu.
```

Next, we run one client on my local machine, as follows:

```
$# Just to show we are running on a different machine

$ hostname
andrew-campbells-macbook-pro.local

$ ./echoClient 129.170.213.32
Hello from andrew-campbells-macbook-pro.local
String received from the server: Hello from andrew-campbells-macbook-pro.local
```

Next, we run one client on my local machine, as follows:

```
$# Just to show we are running on a different machine

$ hostname
andrew-campbells-macbook-pro.local

$ ./echoClient 129.170.213.32
Hello from andrew-campbells-macbook-pro.local
String received from the server: Hello from andrew-campbells-macbook-pro.local
```

Notice, that when we type make a connection and type in "Hello from andrew-campbells-macbook-pro.local" we get the following at the server.

```
$ Received request...
Child created for dealing with client requests
String received from and resent to the client:Hello from andrew-campbells-
macbook-pro.local
```

Now, we ssh into a another machine and start a client

```
$ ssh campbell@moose.cs.dartmouth.edu
campbell@moose.cs.dartmouth.edu's password:
Last login: Mon Feb  8 10:25:01 2010 from 10.35.2.112
$ cd public_html/cs23
$ mygcc -o echoClient echoClient.c
$ ./echoClient 129.170.213.32
Hello from moose.cs.dartmouth.edu
String received from the server: Hello from moose.cs.dartmouth.edu
```

Over at the server we see that the new client is recognized proving that our concurrent server can handle multiple clients at any one time; that is cool!

```
$Received request...
Child created for dealing with client requests
String received from and resent to the client:Hello from moose.cs.dartmouth.edu
```

Elementary UDP sockets: UDP echo server, lack of flow control with UDP

There are some fundamental differences between applications written using TCP versus those that use UDP. These are because of the differences in the two transport layers: UDP is a connectionless, unreliable, datagram protocol, quite unlike the connection-oriented, reliable byte stream provided by TCP. Nevertheless, there are

instances when it makes sense to use UDP instead of TCP, and we will go over this design choice. Some popular applications are built using UDP: DNS, NFS, and SNMP, for example.

Figure shows the function calls for a typical UDP client/server. The client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the sendto function (described in the next section), which requires the address of the destination (the server) as a parameter. Similarly, the server does not accept a connection from a client. Instead, the server just calls the recvfrom function, which waits until data arrives from some client. recvfrom returns the protocol address of the client, along with the datagram, so the server can send a response to the correct client.

Figure: Socket functions for UDP client/server.

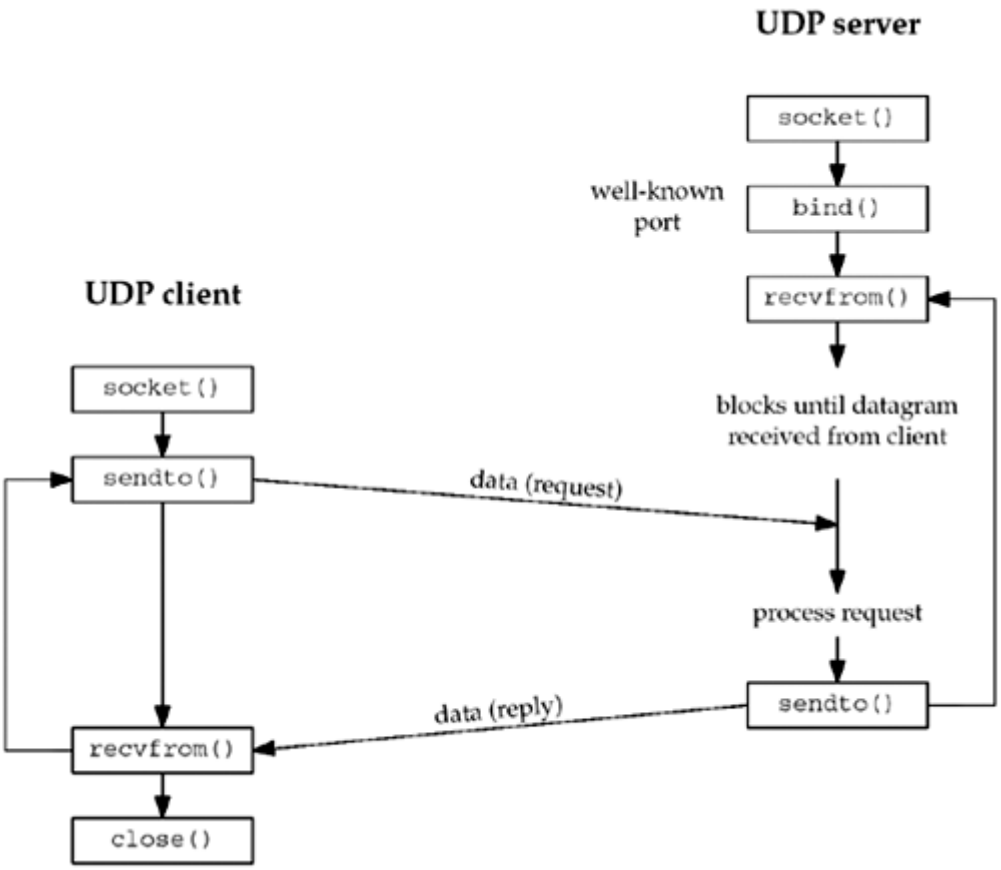


Figure shows a timeline of the typical scenario that takes place for a UDP client/server exchange. We can compare this to the typical TCP exchange.

In this chapter, we will describe the new functions that we use with UDP sockets, recvfrom and sendto, and redo our echo client/server to use UDP. We will also describe the use of the connect function with a UDP socket, and the concept of asynchronous errors.

recvfrom and sendto Functions

These two functions are similar to the standard read and write functions, but three additional arguments are required.

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buff, size_t nbytes, int flags, struct sockaddr *from, socklen_t *addrlen);
```

ssize_t sendto(int *sockfd*, const void **buff*, size_t *nbytes*, int *flags*, const struct sockaddr **to*, socklen_t *addrlen*);
Both return: number of bytes read or written if OK, –1 on error

The first three arguments, *sockfd, buff*, and *nbytes*, are identical to the first three arguments for read and write: descriptor, pointer to buffer to read into or write from, and number of bytes to read or write.

For now, we will always set the *flags* to 0.

The *to* argument for sendto is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is to be sent. The size of this socket address structure is specified by *addrlen*. The recvfrom function fills in the socket address structure pointed to by *from* with the protocol address of who sent the datagram. The number of bytes stored in this socket address structure is also returned to the caller in the integer pointed to by *addrlen*. Note that the final argument to sendto is an integer value, while the final argument to recvfrom is a pointer to an integer value (a value-result argument).

The final two arguments to recvfrom are similar to the final two arguments to accept: The contents of the socket address structure upon return tell us who sent the datagram (in the case of UDP) or who initiated the connection (in the case of TCP). The final two arguments to sendto are similar to the final two arguments to connect: We fill in the socket address structure with the protocol address of where to send the datagram (in the case of UDP) or with whom to establish a connection (in the case of TCP).

Both functions return the length of the data that was read or written as the value of the function. In the typical use of recvfrom, with a datagram protocol, the return value is the amount of user data in the datagram received.

Writing a datagram of length 0 is acceptable. In the case of UDP, this results in an IP datagram containing an IP header (normally 20 bytes for IPv4 and 40 bytes for IPv6), an 8-byte UDP header, and no data. This also means that a return value of 0 from recvfrom is acceptable for a datagram protocol: It does not mean that the peer has closed the connection, as does a return value of 0 from read on a TCP socket. Since UDP is connectionless, there is no such thing as closing a UDP connection.

If the *from* argument to recvfrom is a null pointer, then the corresponding length argument (*addrlen*) must also be a null pointer, and this indicates that we are not interested in knowing the protocol address of who sent us data.

Both recvfrom and sendto can be used with TCP, although there is normally no reason to do so.

UDP Echo Server: main Function

We will now redo our simple echo client/server using UDP. Our UDP client and server programs follow the function call flow that we diagrammed in the Figure. The figure depicts the functions that are used. Figure shows the server main function.

Figure: Simple echo client/server using UDP.

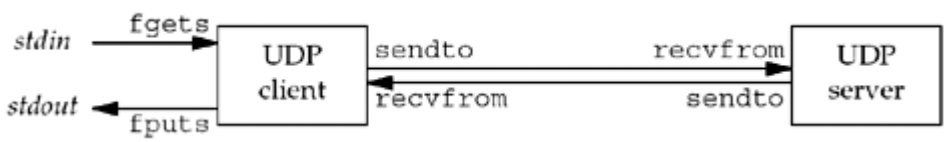


Figure: UDP echo server.

udpcliserv/udpserv01.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5    int    sockfd;
 6    struct sockaddr_in servaddr, cliaddr;

 7    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

 8    bzero(&servaddr, sizeof(servaddr));
 9    servaddr.sin_family = AF_INET;
10    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
11    servaddr.sin_port = htons(SERV_PORT);

12    Bind(sockfd, (SA *) &servaddr, sizeof(servaddr));

13    dg_echo(sockfd, (SA *) &cliaddr, sizeof(cliaddr));
14 }
```

Create UDP socket, bind server's well-known port

7–12 We create a UDP socket by specifying the second argument to socket as SOCK_DGRAM (a datagram socket in the IPv4 protocol). As with the TCP server example, the IPv4 address for the bind is specified as INADDR_ANY and the server's well-known port is the constant SERV_PORT from the unp.h header.

13 The function dg_echo is called to perform server processing.

UDP Echo Server: dg_echo Function

Figure shows the dg_echo function.

Figure 8.4 dg_echo function: echo lines on a datagram socket.

lib/dg_echo.c

```
 1 #include    "unp.h"

 2 void
 3 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 4 {
 5    int    n;
 6    socklen_t len;
 7    char    mesg[MAXLINE];

 8    for ( ; ; ) {
 9       len = clilen;
10       n = Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);
```

```
11        Sendto(sockfd, mesg, n, 0, pcliaddr, len);
12    }
13 }
```

Read datagram, echo back to sender

8–12 This function is a simple loop that reads the next datagram arriving at the server's port using recvfrom and sends it back using sendto.

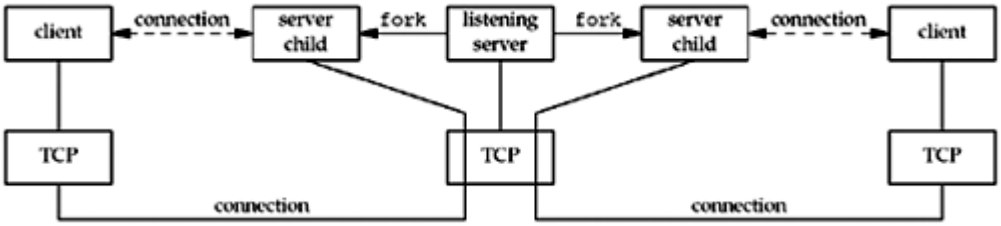
Despite the simplicity of this function, there are numerous details to consider. First, this function never terminates. Since UDP is a connectionless protocol, there is nothing like an EOF as we have with TCP.

Next, this function provides an *iterative server*, not a concurrent server as we had with TCP. There is no call to fork, so a single server process handles any and all clients. In general, most TCP servers are concurrent and most UDP servers are iterative.

There is implied queuing taking place in the UDP layer for this socket. Indeed, each UDP socket has a receive buffer and each datagram that arrives for this socket is placed in that socket receive buffer. When the process calls recvfrom, the next datagram from the buffer is returned to the process in a first-in, first-out (FIFO) order. This way, if multiple datagrams arrive for the socket before the process can read what's already queued for the socket, the arriving datagrams are just added to the socket receive buffer. But, this buffer has a limited size. We discussed this size and how to increase it with the SO_RCVBUF socket option.

Figure summarizes our TCP client/server when two clients establish connections with the server.

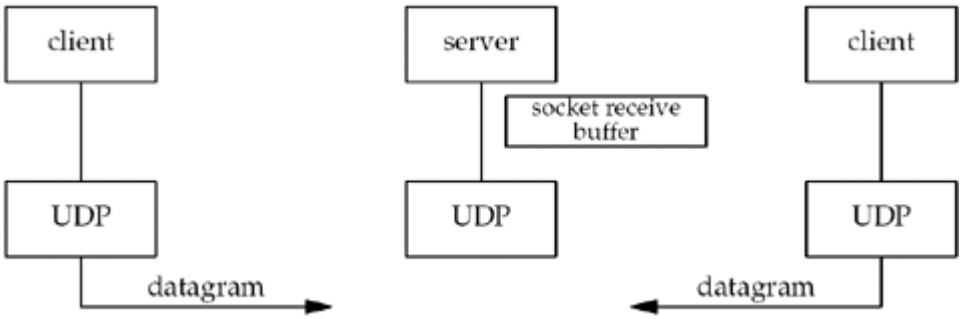
Figure: Summary of TCP client/server with two clients.



There are two connected sockets and each of the two connected sockets on the server host has its own socket receive buffer.

Figure shows the scenario when two clients send datagrams to our UDP server.

Figure: Summary of UDP client/server with two clients.



There is only one server process and it has a single socket on which it receives all arriving datagrams and sends all responses. That socket has a receive buffer into which all arriving datagrams are placed.

The main function in Figure is *protocol-dependent* (it creates a socket of protocol AF_INET and allocates and initializes an IPv4 socket address structure), but the dg_echo function is *protocol-independent*. The reason dg_echo is protocol-independent is because the caller (the main function in our case) must allocate a socket address structure of the correct size, and a pointer to this structure, along with its size, are passed as arguments to dg_echo. The function dg_echo never looks inside this protocol-dependent structure: It simply passes a pointer to the structure to recvfrom and sendto. recvfrom fills this structure with the IP address and port number of the client, and since the same pointer (pcliaddr) is then passed to sendto as the destination address, this is how the datagram is echoed back to the client that sent the datagram.

UDP Echo Client: main Function

The UDP client main function is shown in Figure.

Figure 8.7 UDP echo client.

udpcliserv/udpcli01.c

```
 1 #include    "unp.h"

 2 int
 3 main(int argc, char **argv)
 4 {
 5    int    sockfd;
 6    struct sockaddr_in servaddr;

 7    if(argc != 2)
 8      err_quit("usage: udpcli <IPaddress>");

 9    bzero(&servaddr, sizeof(servaddr));
10    servaddr.sin_family = AF_INET;
11    servaddr.sin_port = htons(SERV_PORT);
12    Inet_pton(AF_INET, argv[1], &servaddr.sin_addr);

13    sockfd = Socket(AF_INET, SOCK_DGRAM, 0);

14    dg_cli(stdin, sockfd, (SA *) &servaddr, sizeof(servaddr));

15    exit(0);
16 }
```

Fill in socket address structure with server's address

9–12 An IPv4 socket address structure is filled in with the IP address and port number of the server. This structure will be passed to dg_cli, specifying where to send datagrams.

13–14 A UDP socket is created and the function dg_cli is called.

UDP Echo Client: dg_cli Function

Figure shows the function dg_cli, which performs most of the client processing.

Figure: dg_cli function: client processing loop.

lib/dg_cli.c

```
 1 #include    "unp.h"

 2 void
 3 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 4 {
 5    int    n;
 6    char   sendline[MAXLINE], recvline[MAXLINE + 1];

 7    while (Fgets(sendline, MAXLINE, fp) != NULL) {

 8       Sendto(sockfd, sendline, strlen(sendline), 0, pservaddr, servlen);

 9       n = Recvfrom(sockfd, recvline, MAXLINE, 0, NULL, NULL);

10       recvline[n] = 0;       /* null terminate */
11       Fputs(recvline, stdout);
12    }
13 }
```

7–12 There are four steps in the client processing loop: read a line from standard input using fgets, send the line to the server using sendto, read back the server's echo using recvfrom, and print the echoed line to standard output using fputs.

Our client has not asked the kernel to assign an ephemeral port to its socket. (With a TCP client, we said the call to connect is where this takes place.) With a UDP socket, the first time the process calls sendto, if the socket has not yet had a local port bound to it, that is when an ephemeral port is chosen by the kernel for the socket. As with TCP, the client can call bind explicitly, but this is rarely done.

Notice that the call to recvfrom specifies a null pointer as the fifth and sixth arguments. This tells the kernel that we are not interested in knowing who sent the reply. There is a risk that any process, on either the same host or some other host, can send a datagram to the client's IP address and port, and that datagram will be read by the client, who will think it is the server's reply.

As with the server function dg_echo, the client function dg_cli is protocol-independent, but the client main function is protocol-dependent. The main function allocates and initializes a socket address structure of some protocol type and then passes a pointer to this structure, along with its size, to dg_cli.

Lost Datagrams

Our UDP client/server example is not reliable. If a client datagram is lost (say it is discarded by some router between the client and server), the client will block forever in its call to recvfrom in the function dg_cli, waiting for a server reply that will never arrive. Similarly, if the client datagram arrives at the server but the server's reply is lost, the client will again block forever in its call to recvfrom. A typical way to prevent this is to place a timeout on the client's call to recvfrom.

Just placing a timeout on the recvfrom is not the entire solution. For example, if we do time out, we cannot tell whether our datagram never made it to the server, or if the server's reply never made it back. If the client's request was something like "transfer a certain amount of money from account A to account B" (instead of our simple echo server), it would make a big difference as to whether the request was lost or the reply was lost.

Server Not Running

The next scenario to examine is starting the client without starting the server. If we do so and type in a single line to the client, nothing happens. The client blocks forever in its call to recvfrom, waiting for a server reply that will never appear. But, this is an example where we need to understand more about the underlying protocols to understand what is happening to our networking application.

First we start tcpdump on the host macosx, and then we start the client on the same host, specifying the host freebsd4 as the server host. We then type a single line, but the line is not echoed.

```
macosx % udpcli01 172.24.37.94
hello, world
we type this line but nothing is echoed back
```

Figure shows the tcpdump output.

Figure tcpdump output when server process not started on server host.

```
1 0.0               arp who-has freebsd4 tell macosx
2 0.003576 ( 0.0036)    arp reply freebsd4 is-at 0:40:5:42:d6:de

3 0.003601 ( 0.0000)    macosx.51139 > freebsd4.9877: udp 13
4 0.009781 ( 0.0062)    freebsd4 > macosx: icmp: freebsd4 udp port 9877 unreachable
```

First we notice that an ARP request and reply are needed before the client host can send the UDP datagram to the server host. (We left this exchange in the output to reiterate the potential for an ARP request-reply before an IP datagram can be sent to another host or router on the local network.)

In line 3, we see the client datagram sent but the server host responds in line 4 with an ICMP "port unreachable." (The length of 13 accounts for the 12 characters and the newline.) This ICMP error, however, is not returned to the client process, for reasons that we will describe shortly. Instead, the client blocks forever in the call to recvfrom in Figure. We also note that ICMPv6 has a "port unreachable" error, similar to ICMPv4, so the results described here are similar for IPv6.

We call this ICMP error an *asynchronous error*. The error was caused by sendto, but sendto returned successfully. Recall that a successful return from a UDP output operation only means there was room for the resulting IP datagram on the interface output queue. The ICMP error is not returned until later, which is why it is called asynchronous.

The basic rule is that an asynchronous error is not returned for a UDP socket unless the socket has been connected. Why this design decision was made when sockets were first implemented is rarely understood. (The implementation implications are discussed on pp. 748–749 of TCPv2.)

Consider a UDP client that sends three datagrams in a row to three different servers (i.e., three different IP addresses) on a single UDP socket. The client then enters a loop that calls recvfrom to read the replies. Two of the datagrams are correctly delivered (that is, the server was running on two of the three hosts) but the third host was not running the server. This third host responds with an ICMP port unreachable. This ICMP error message contains the IP header and UDP header of the datagram that caused the error. (ICMPv4 and ICMPv6 error messages always contain the IP header and all of the UDP header or part of the TCP header to allow the receiver of the ICMP error to determine which socket caused the error. The client that sent the three datagrams needs to know the destination of the datagram that caused the error to distinguish which of the three datagrams

caused the error. But how can the kernel return this information to the process? The only piece of information that recvfrom can return is an errno value; recvfrom has no way of returning the destination IP address and destination UDP port number of the datagram in error. The decision was made, therefore, to return these asynchronous errors to the process only if the process connected the UDP socket to exactly one peer.

Linux returns most ICMP "destination unreachable" errors even for unconnected sockets, as long as the SO_BSDCOMPAT socket option is not enabled. All the ICMP "destination unreachable" errors are returned, except codes 0, 1, 4, 5, 11, and 12.

We return to this problem of asynchronous errors with UDP sockets and show an easy way to obtain these errors on unconnected sockets using a daemon of our own.

Lack of Flow Control with UDP

We now examine the effect of UDP not having any flow control. First, we modify our dg_cli function to send a fixed number of datagrams. It no longer reads from standard input. This function writes 2,000 1,400-byte UDP datagrams to the server.

We next modify the server to receive datagrams and count the number received. This server no longer echoes datagrams back to the client. When we terminate the server with our terminal interrupt key (SIGINT), it prints the number of received datagrams and terminates.

dg_cli function that writes a fixed number of datagrams to the server.

udpcliserv/dgcliloop1.c

```
 1 #include    "unp.h"

 2 #define NDG      2000        /*  datagrams to send */
 3 #define DGLEN    1400         /*  length of each datagram */

 4 void
 5 dg_cli(FILE *fp, int sockfd, const SA *pservaddr, socklen_t servlen)
 6 {
 7    int    i;
 8    char   sendline[DGLEN];

 9    for (i = 0; i < NDG; i++) {
10       Sendto(sockfd, sendline, DGLEN, 0, pservaddr, servlen);
11    }
12 }
```

Figure: dg_echo function that counts received datagrams.

udpcliserv/dgecholoop1.c

```
 1 #include    "unp.h"

 2 static void recvfrom_int(int);
 3 static int count;

 4 void
 5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 6 {
 7    socklen_t len;
```

```
 8    char    mesg[MAXLINE];

 9    Signal(SIGINT, recvfrom_int);

10    for ( ; ; ) {
11        len = clilen;
12        Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

13        count++;
14    }
15 }

16 static void
17 recvfrom_int(int signo)
18 {
19    printf("\nreceived %d datagrams\n", count);
20    exit(0);
21 }
```

We now run the server on the host freebsd, a slow SPARCStation. We run the client on the RS/6000 system aix, connected directly with 100Mbps Ethernet. Additionally, we run netstat -s on the server, both before and after, as the statistics that are output tell us how many datagrams were lost. Figure shows the output on the server.

Figure: Output on server host.

```
freebsd % netstat -s -p udp
udp:
        71208 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        0 with no checksum
        832 dropped due to no socket
        16 broadcast/multicast datagrams dropped due to no socket
        1971 dropped due to full socket buffers
        0 not for hashed pcb
        68389 delivered
        137685 datagrams output
freebsd % udpserv06          start our server
                                                we run the client here
    ^C                      we type our interrupt key after the client is finished
received 30 datagrams
freebsd % netstat -s -p udp
udp:
        73208 datagrams received
        0 with incomplete header
        0 with bad data length field
        0 with bad checksum
        0 with no checksum
        832 dropped due to no socket
        16 broadcast/multicast datagrams dropped due to no socket
        3941 dropped due to full socket buffers
        0 not for hashed pcb
        68419 delivered
        137685 datagrams output
```

The client sent 2,000 datagrams, but the server application received only 30 of these, for a 98% loss rate. There is *no* indication whatsoever to the server application or to the client application that these datagrams were lost. As we have said, UDP has no flow control and it is unreliable. It is trivial, as we have shown, for a UDP sender to overrun the receiver.

If we look at the netstat output, the total number of datagrams received by the server host (not the server application) is 2,000 (73,208 - 71,208). The counter "dropped due to full socket buffers" indicates how many datagrams were received by UDP but were discarded because the receiving socket's receive queue was full (p. 775 of TCPv2). This value is 1,970 (3,491 - 1,971), which when added to the counter output by the application (30), equals the 2,000 datagrams received by the host. Unfortunately, the netstat counter of the number dropped due to a full socket buffer is systemwide. There is no way to determine which applications (e.g., which UDP ports) are affected.

The number of datagrams received by the server in this example is not predictable. It depends on many factors, such as the network load, the processing load on the client host, and the processing load on the server host.

If we run the same client and server, but this time with the client on the slow Sun and the server on the faster RS/6000, no datagrams are lost.

```
aix % udpserv06
^?                              we type our interrupt key after the client is finished
received 2000 datagrams
```

UDP Socket Receive Buffer

The number of UDP datagrams that are queued by UDP for a given socket is limited by the size of that socket's receive buffer. We can change this with the SO_RCVBUF socket option. The default size of the UDP socket receive buffer under FreeBSD is 42,080 bytes, which allows room for only 30 of our 1,400-byte datagrams. If we increase the size of the socket receive buffer, we expect the server to receive additional datagrams. Figure shows a modification to the dg_echo function from Figure that sets the socket receive buffer to 240 KB.

Figure: dg_echo function that increases the size of the socket receive queue.

udpcliserv/dgecholoop2.c

```
 1 #include   "unp.h"

 2 static void recvfrom_int(int);
 3 static int count;

 4 void
 5 dg_echo(int sockfd, SA *pcliaddr, socklen_t clilen)
 6 {
 7    int     n;
 8    socklen_t len;
 9    char    mesg[MAXLINE];

10    Signal(SIGINT, recvfrom_int);

11    n = 220 * 1024;
12    Setsockopt(sockfd, SOL_SOCKET, SO_RCVBUF, &n, sizeof(n));

13    for ( ; ; ) {
```

```
14          len = clilen;
15          Recvfrom(sockfd, mesg, MAXLINE, 0, pcliaddr, &len);

16          count++;
17      }
18  }

19  static void
20  recvfrom_int(int signo)
21  {
22      printf("\nreceived %d datagrams\n", count);
23      exit(0);
24  }
```

If we run this server on the Sun and the client on the RS/6000, the count of received datagrams is now 103. While this is slightly better than the earlier example with the default socket receive buffer, it is no panacea.

Why do we set the receive socket buffer size to 220 x 1,024 in figure? The maximum size of a socket receive buffer in FreeBSD 5.1 defaults to 262,144 bytes (256 x 1,024), but due to the buffer allocation policy (described in Chapter 2 of TCPv2), the actual limit is 233,016 bytes. Many earlier systems based on 4.3BSD restricted the size of a socket buffer to around 52,000 bytes.