CS 60 Computer Networks

Lecture 3 and 4

Socket Programming

How do we build Internet applications? In this lecture, we will discuss the socket API and support for TCP and UDP communications between end hosts. Socket programing is the key API for programming distributed applications on the Internet.

BTW, Kurose/Ross only cover Java socket programming and not C socket programming discussed below.

Goals

We plan to learn the following from these lectures:

- What is a socket?
- The client-server model
- Byte order
- TCP socket API
- UDP socket API
- Concurrent server design

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 ad 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

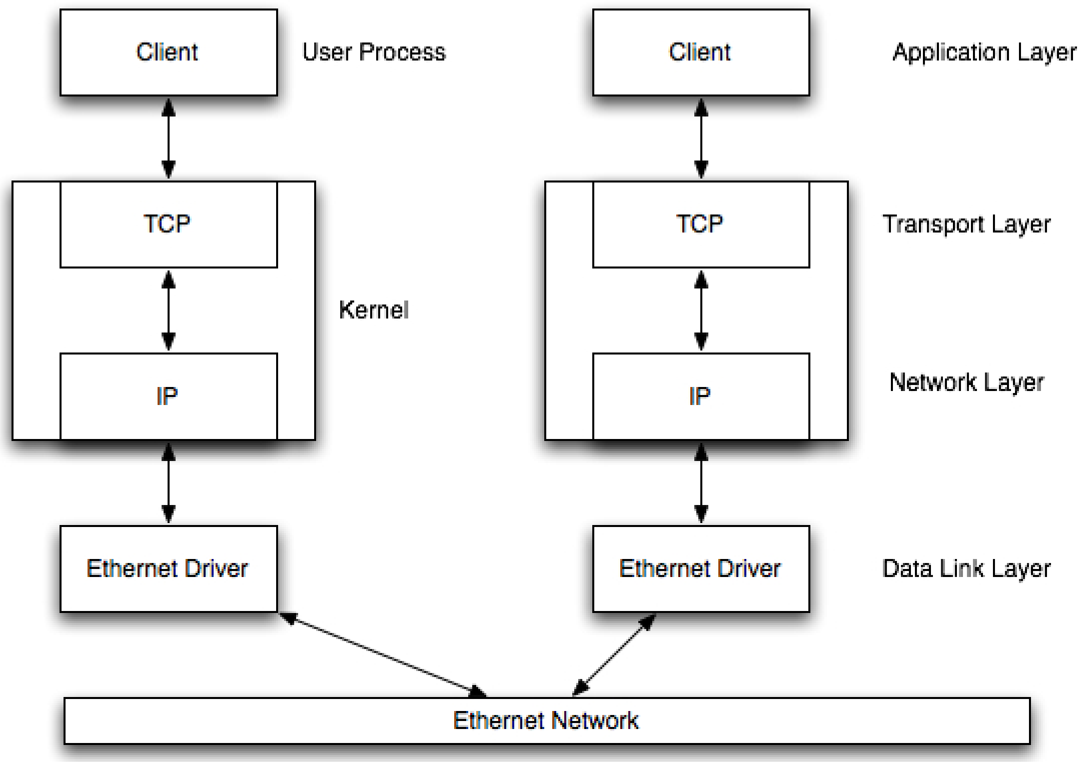

Figure 1: Client and server on the same Ethernet communicating using TCP/IP.

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.

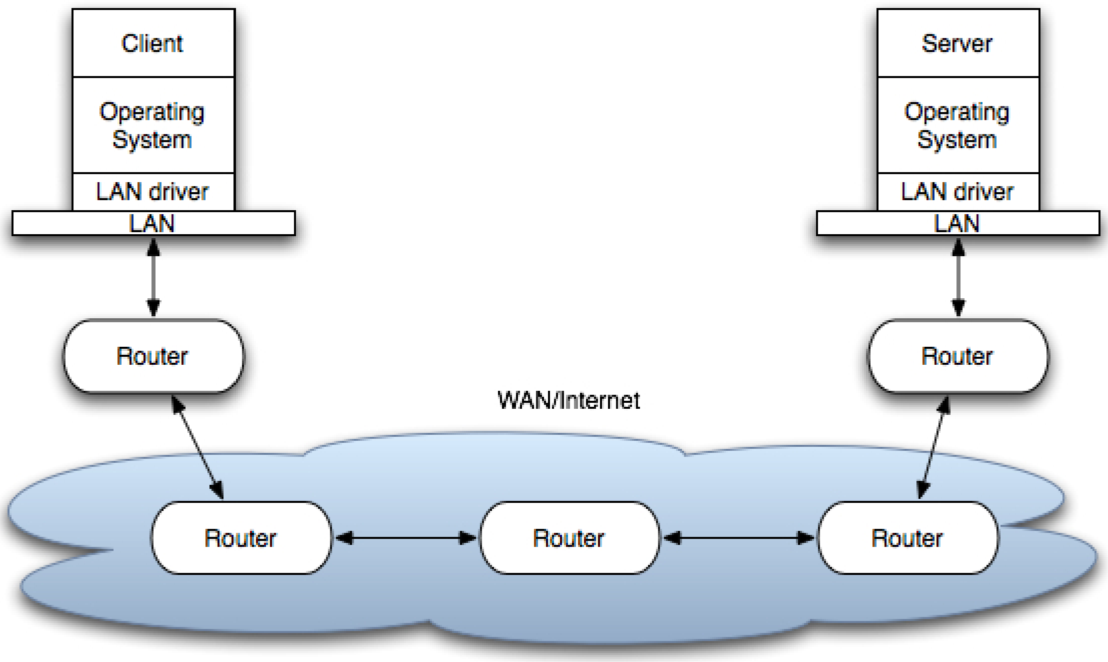

Figure 2: Client and server on different LANs connected through WAN/Internet.

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.

User Datagram Protocol (UDP)

UDP is a simple transport-layer protocol. The application writes a message to a UDP socket, which is then encapsulated in a UDP datagram, which is further encapsulated in an IP datagram, which is sent to the destination.

There is no guarantee that a UDP will reach the destination, that the order of the datagrams will be preserved across the network or that datagrams arrive only once.

The problem of UDP is its lack of reliability: if a datagram reaches its final destination but the checksum detects an error, or if the datagram is dropped in the network, it is not automatically retransmitted.

Each UDP datagram is characterized by a length. The length of a datagram is passed to the receiving application along with the data.

No connection is established between the client and the server and, for this reason, we say that UDP provides a *connection-less service*.

It is described in RFC 768.

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.

We have mentioned that UDP datagrams are characterized by a length. 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 and 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.

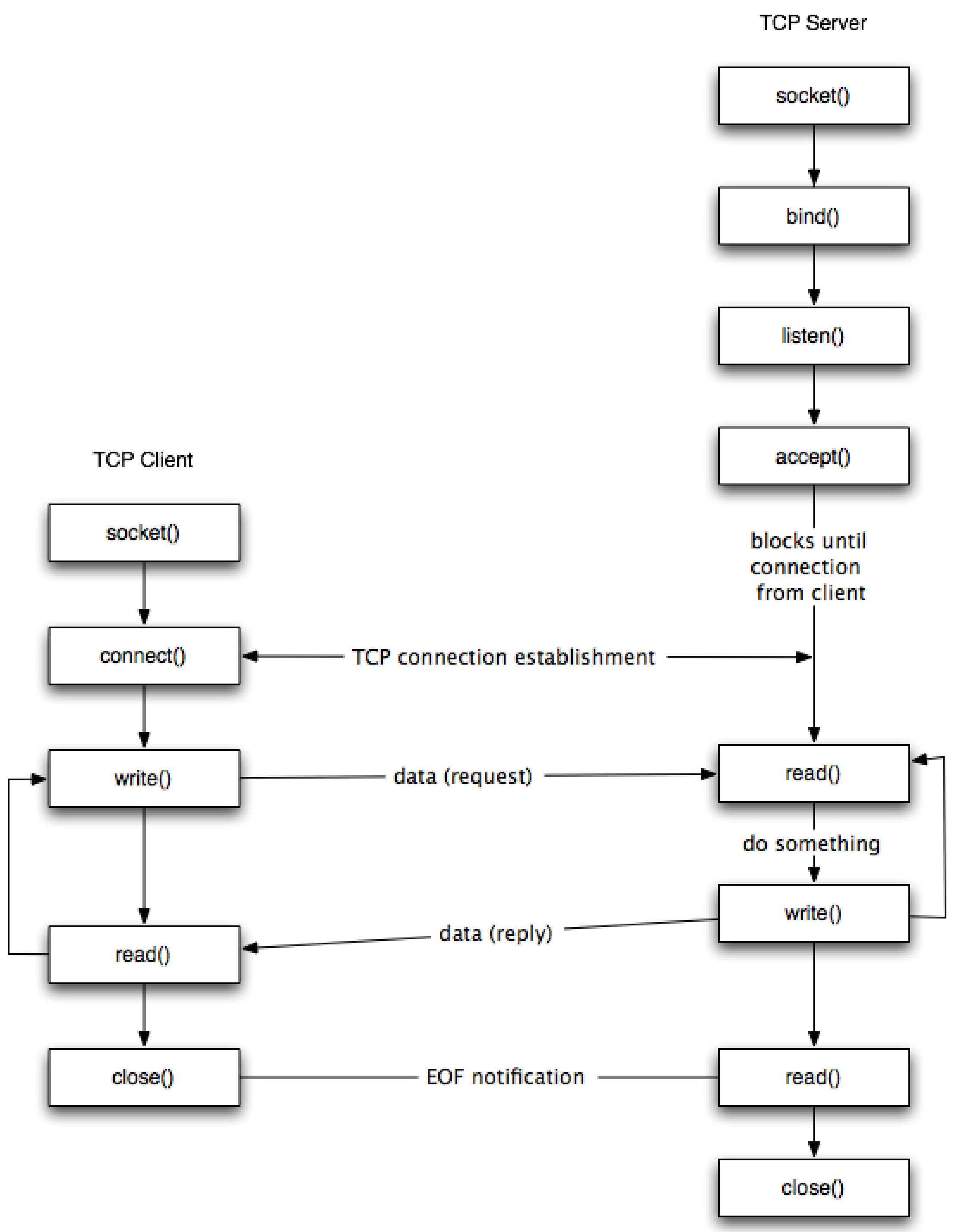

Figure 3: TCP client-server.

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.
- Close the connection by means of the `close()` function.

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()`.
- Close the connection by means of the `close()` function.

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, `servaddr` 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

Note, the local host address is 127.0.0.1; for example, if you wanted to run your echoServer (see later) on your local machine the your client would connect to 127.0.0.1 with the suitable 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 it should enqueue on 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 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);
```

UDP Socket API

There are some fundamental differences between TCP and UDP sockets. UDP is a connection-less, unreliable, datagram protocol (TCP is instead connection-oriented, reliable and stream based). There are some instances when it makes to use UDP instead of TCP. Some popular applications built around UDP are DNS, NFS, SNMP and for example, some Skype services and streaming media.

Figure 4 shows the the interaction between a UDP client and server. First of all, the client does not establish a connection with the server. Instead, the client just sends a datagram to the server using the `sendto` function which requires the address of the destination 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 IP address of the client, along with the datagram, so the server can send a response to the client.

As shown in the Figure, the steps of establishing a UDP socket communication on the client side are as follows:

- Create a socket using the `socket()` function;
- Send and receive data by means of the `recvfrom()` and `sendto()` functions.

The steps of establishing a UDP socket communication on the server side are as follows:

- Create a socket with the `socket()` function;
- Bind the socket to an address using the `bind()` function;
- Send and receive data by means of `recvfrom()` and `sendto()`.

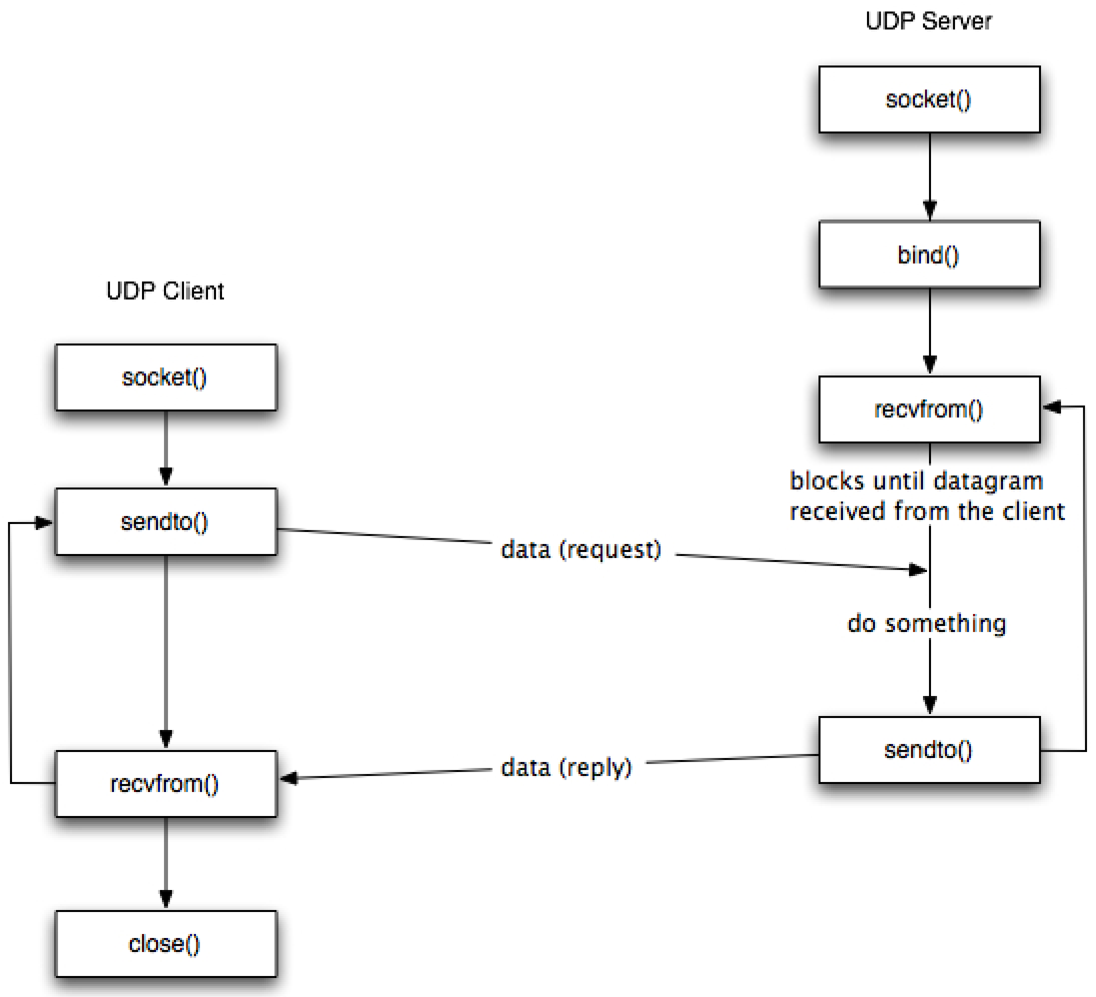

Figure 4: UDP client-server.

In this section, we will describe the two new functions `recvfrom()` and `sendto()`.

The recvfrom() Function

This function is similar to the `read()` function, but three additional arguments are required. The `recvfrom()` function is defined as follows:

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

ssize_t recvfrom(int sockfd, void* buff, size_t nbytes,
                           int flags, struct sockaddr* from,
                           socklen_t *addrlen);
```

The first three arguments `sockfd`, `buff`, and `nbytes`, are identical to the first three arguments of `read` and `write`. `sockfd` is the socket descriptor, `buff` is the pointer to read into, and `nbytes` is number of bytes to read. In our examples we will set all the values of the `flags` argument to 0. 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 the socket address structure is returned in the integer pointed by `addrlen`.

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

The sendto() Function

This function is similar to the `send()` function, but three additional arguments are required. The `sendto()` function is defined as follows:

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

The first three arguments `sockfd`, `buff`, and `nbytes`, are identical to the first three arguments of `recv`. `sockfd` is the socket descriptor, `buff` is the pointer to write from, and `nbytes` is number of bytes to write. In our examples we will set all the values of the `flags` argument to 0. The `to` argument is a socket address structure containing the protocol address (e.g., IP address and port number) of where the data is sent. `addlen` specified the size of this socket.

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

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). We do not consider this kind of servers in this course.

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 5.

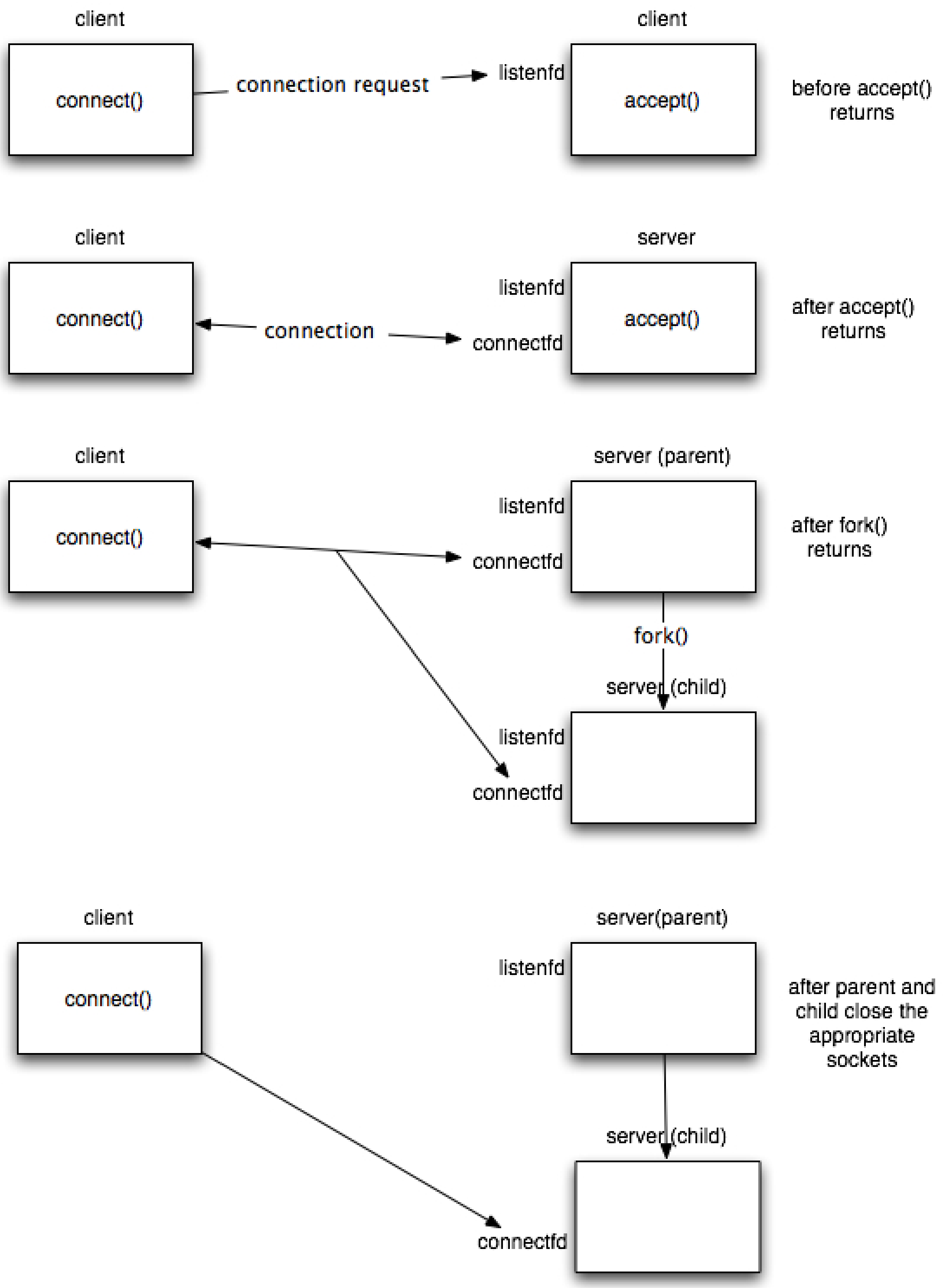

Figure 5: Example of interaction among a client and a concurrent server.

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.

echoClient.c source: echoClient.c

TCP Echo Client

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

echoServer.c source: echoServer.c

TCP Iterative Server

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <string.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);
}
```

conEchoServer.c source: conEchoServer.c

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>

#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);
}
}
```