Network Programming

Topics

• Client/Server model.
• Berkeley sockets:
  – TCP client and server examples.
  – UDP client and server examples.
• I/O multiplexing with `select()`.
Favourite Reference

• Excellent tutorials.
• Empathetic man pages.
• Great starting point.
Internet protocol stack

Packet encapsulation:

- Ethernet
- IP
- TCP
- FTP
- Data
UDP vs TCP

**User Datagram Protocol (UDP)**

- Unreliable datagrams from process to process.
- Thin wrapper layer on top of IP.
- Similar to sending surface mail:
  - Each message is an independent chunk of data (datagram).
  - Messages may not arrive or may arrive out of order.
- Faster than TCP, requires no server state, but unreliable.
- Tunneling/VPN, media streaming, VoIP (Skype), games, etc.

**Transmission Control Protocol (TCP)**

- Reliable byte-stream from process to process.
- Complex implementation.
- Similar to placing a phone call:
  - No messages; a continuous stream of bytes over a connection.
  - Bytes arrive in order.
- Slower and requires more resources, but cleaner user semantics.
- HTTP, SSH, FTP, telnet, SMTP, POP/IMAP, etc.
Berkeley Sockets Interface

Created in the early 80’s as part of the original Berkeley distribution of Unix that contained the TCP/IP protocol stack.

Provides user-level interface to:
- TCP ("SOCK_STREAM")
- UDP ("SOCK_DGRAM")

Underlying basis for all Internet applications.

Based on client/server programming model.
What is a socket?

A *socket* is a descriptor that lets an application read/write from/to the network:

- Unix uses the same abstraction for both file I/O and network I/O.
- Socket is a *communication endpoint*.

Clients and servers communicate with each other via TCP and UDP using the same socket abstraction:

- Applications read and write TCP byte streams by reading from and writing to socket descriptors.
- Applications read / write UDP datagrams by reading from and writing to socket descriptors.

Main difference between file I/O and socket I/O is how the application “opens” the socket descriptors.
Key data structures

Defined in `/usr/include/netinet/in.h`:

```c
/* Internet address */
typedef uint32_t in_addr_t;
struct in_addr {
    in_addr_t s_addr; /* 32-bit IP address */
};

/* Internet style socket address */
struct sockaddr_in {
    unsigned short int sin_family;/* Address family (AF_INET) */
    unsigned short int sin_port; /* Port number */
    struct in_addr sin_addr; /* IP address */
    unsigned char sin_zero[...]; /* Pad to sizeof “struct sockaddr”*/
};
```

Internet-style sockets are characterized by a 32-bit IP address and a 16-bit port number.
Key data structures

Defined in /usr/include/netdb.h:

```c
/* Domain Name Service (DNS) host entry */
struct hostent {
    char    *h_name;        /* official name of host */
    char    **h_aliases;    /* alias list */
    int     h_addrtype;     /* host address type */
    int     h_length;       /* length of address */
    char    **h_addr_list;  /* list of addresses */
}
```

`Hostent` is a DNS host entry that associates a domain name (e.g., cmu.edu) with an IP address (128.2.35.186):

- DNS is a world-wide distributed database of domain name/IP address mappings.
- Can be accessed from user programs using `gethostbyname()` [domain name to IP address] or `gethostbyaddr()` [IP address to domain name].
- Can also be accessed from the shell using `nslookup` or `dig`. 
TCP echo server: prologue

The server listens on a port passed via the command line.

```c
/*
 * error - wrapper for perror
 */
void error(char *msg) {
    perror(msg);
    exit(1);
}

int main(int argc, char **argv) {
    /* local variable definitions */

    /*
     * check command line arguments
     */
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(1);
    }
    portno = atoi(argv[1]);
    ...
}
```
TCP echo server: socket()

socket() creates a parent socket (a listening socket).

```c
int parentfd; /* parent socket descriptor */
parentfd = socket(AF_INET, SOCK_STREAM, 0);
if (parentfd < 0)
    error("ERROR opening socket");
```

socket() returns an integer (socket descriptor):
- `parentfd < 0` indicates that an error occurred.

AF_INET: indicates that the socket is associated with Internet protocols.

SOCK_STREAM: selects the TCP protocol.
TCP echo server: setsockopt()

The socket can be given some attributes.

```c
optval = 1;
setsockopt(parentfd, SOL_SOCKET, SO_REUSEADDR,
           (const void *)&optval, sizeof(int));
```

Handy trick that allows us to rerun the server immediately after we kill it:

- Otherwise, would have to wait about 15 secs.
- Eliminates the “Address already in use” error.
- Suggest you do this for all your servers.
TCP echo server: init socket address

Next, we initialize the socket attributes with the server’s Internet address (IP address and port).

```c
struct sockaddr_in serveraddr; /* server's addr */
/* this is an Internet address */
memset((char *) &serveraddr, 0, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;

/* a client can connect to any of my IP addresses */
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);

/* this is the port to associate the socket with */
serveraddr.sin_port = htons((unsigned short)portno);
```
Network vs. Host Byte Order

Ports and addresses specified in function calls related to sockets must use the network byte order convention:

- Defined as *big-endian*.
- Hosts using different architectures can exchange information without confusion.

The following C functions convert values between host and network byte order:

- `htonl()`: converts `uint32_t` from host byte order to network byte order.
- `htons()`: converts `uint16_t` from host byte order to network byte order.
- `ntohl()`: converts `uint32_t` from network byte order to host byte order.
- `ntohs()`: converts `uint16_t` from network byte order to host byte order.
TCP echo server: bind()

bind() associates the socket with a port:

- Optionally with an IP address if the host has multiple network interfaces.

```c
int parentfd; /* parent socket */
struct sockaddr_in serveraddr; /* server's addr */

/*
 * An invocation to socket() must take place first
 * and the parent socket must be properly configured.
 */
if (bind(parentfd, (struct sockaddr *) &serveraddr,
        sizeof(serveraddr)) < 0)
    error("ERROR on binding");
```
TCP echo server: listen()

`listen()` indicates that this socket will accept TCP connection requests from clients.

```c
int parentfd;                /* parent socket */
if (listen(parentfd, 5) < 0) /* allow 5 requests to queue up */
    error("ERROR on listen");
```

We’re finally ready to enter the main server loop that accepts and processes client connection requests.
TCP echo server: main loop

The server loops endlessly, waiting for connection requests, then reading input from the client, and echoing the input back to the client.

```c
int main(int argc, char *argv[]) {
    /* create and configure the socket */
    while(1) {
        /* accept(): wait for a connection request */
        /* read(): read an input line from the client */
        /* write(): echo the line back to the client */
        /* close(): close the connection */
    }
    return 0;
}
```
TCP echo server: accept()

`accept()` blocks waiting for a connection request.

```c
int parentfd;    /* parent socket */
int childfd;     /* child socket */
int clientlen;   /* byte size of client's address */
struct sockaddr_in clientaddr; /* client addr */

clientlen = sizeof(clientaddr);
childfd = accept(parentfd, (struct sockaddr *) &clientaddr, &clientlen);
if (childfd < 0)
  error("ERROR on accept");
```

`accept()` returns a child socket (`connected` socket) descriptor (`childfd`) with the same properties as `parentfd`:

- Useful for concurrent servers where the parent forks off a process for each connection request.
- All I/O with the client will be done via the child socket.

`accept()` also fills in client’s address.
TCP echo server: identifying client

The server can determine the domain name and IP address of the client.

```c
struct sockaddr_in clientaddr; /* client addr */
struct hostent *hostp;         /* client DNS host entry */
char *hostaddrp;               /* dotted decimal host addr string */

hostp = gethostbyaddr((const char *)&clientaddr.sin_addr.s_addr,
                      sizeof(clientaddr.sin_addr.s_addr), AF_INET);
if (hostp == NULL)
    error("ERROR on gethostbyaddr");

hostaddrp = inet_ntoa(clientaddr.sin_addr);
if (hostaddrp == NULL)
    error("ERROR on inet_ntoa\n");

printf("server established connection with %s (%s)\n",
       hostp->h_name, hostaddrp);
```
TCP echo server: read()

The server reads an ASCII input line from the client.

```
int childfd;       /* child socket */
char buf[BUFSIZE]; /* message buffer */
int n;             /* message byte size */

memset(buf, 0, BUFSIZE);
n = read(childfd, buf, BUFSIZE);
if (n < 0)
    error("ERROR reading from socket");
printf("server received %d bytes: %s", n, buf);
```

At this point, it looks just like file I/O.

What are the possible return values of `read`?
TCP echo server: write()

Finally, the server echoes the input line back to the client, closes the connection, and loops back to wait for the next connection request.

```c
int childfd;       /* child socket */
char buf[BUFSIZE]; /* message buffer */
int n;             /* message byte size */

n = write(childfd, buf, strlen(buf));
if (n < 0)
    error("ERROR writing to socket");

close(childfd);
```

What are the possible return values of `write`?

- **EPIPE** `fd` is connected to a pipe or socket whose reading end is closed. When this happens the writing process will also receive a SIGPIPE signal. (Thus, the write return value is seen only if the program catches, blocks or ignores this signal.)
Testing the TCP server with telnet

bass> tcpserver 5000
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 5 bytes: 123
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 8 bytes: 456789

kittyhawk> telnet bass 5000
Trying 128.2.222.85...
Connected to BASS.CMCL.CS.CMU.EDU.
Escape character is '^]'.
123
123
Connection closed by foreign host.
kittyhawk> telnet bass 5000
Trying 128.2.222.85...
Connected to BASS.CMCL.CS.CMU.EDU.
Escape character is '^]'.
456789
456789
Connection closed by foreign host.
kittyhawk>
TCP client: prologue

The client connects to a host and port passed in on the command line.

```c
/*
 * error - wrapper for perror
 */
void error(char *msg) {
    perror(msg);
    exit(1);
}

int main(int argc, char **argv) {
    /* local variable definitions */
    /* check command line arguments */
    /* check command line arguments */
    if (argc != 3) {
        fprintf(stderr,"usage: %s <hostname> <port>\n", argv[0]);
        exit(1);
    }
    hostname = argv[1];
    portno = atoi(argv[2]);
    ...
}
```
TCP client: socket()

The client creates a socket.

```c
int sockfd; /* socket descriptor */
sockfd = socket(AF_INET, SOCK_STREAM, 0);
if (sockfd < 0)
  error("ERROR opening socket");
```
TCP client: gethostbyname()

The client builds the server’s Internet address.

```c
struct sockaddr_in serveraddr; /* server address */
struct hostent *server;        /* server DNS host entry */
char *hostname;                /* server domain name */

/* gethostbyname: get the server's DNS entry */
server = gethostbyname(hostname);
if (server == NULL) {
    fprintf(stderr,"ERROR, no such host as %s\n", hostname);
    exit(1);
}

/* build the server's Internet address */
bzero((char *) &serveraddr, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
bcopy((char *)server->h_addr, (char *)&serveraddr.sin_addr.s_addr, server->h_length);
serveraddr.sin_port = htons(portno);
```
TCP client: connect()

Then the client creates a connection with the server.

```c
int sockfd; /* socket descriptor */
struct sockaddr_in serveraddr; /* server address */

if (connect(sockfd, &serveraddr, sizeof(serveraddr)) < 0)
    error("ERROR connecting");
```

At this point, the client is ready to begin exchanging messages with the server via `sockfd`:

- Notice that there is no notion of a parent and child socket on a client.
TCP client: read(), write(), close()

The client reads a message from stdin, sends it to the server, waits for the echo, and terminates.

```c
/* get message line from the user */
printf("Please enter msg: ");
memset(buf, 0, BUFSIZE);
fgets(buf, BUFSIZE, stdin);

/* send the message line to the server */
n = write(sockfd, buf, strlen(buf));
if (n < 0)
    error("ERROR writing to socket");

/* print the server's reply */
memset(buf, 0, BUFSIZE);
n = read(sockfd, buf, BUFSIZE);
if (n < 0)
    error("ERROR reading from socket");

printf("Echo from server: %s", buf);
close(sockfd);
return 0;
```
Running the TCP client and server

bass> tcpserver 5000
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 4 bytes: 123
server established connection with KITTYHAWK.CMCL (128.2.194.242)
server received 7 bytes: 456789
...

kittyhawk> tcpclient bass 5000
Please enter msg: 123
Echo from server: 123
kittyhawk> tcpclient bass 5000
Please enter msg: 456789
Echo from server: 456789
kittyhawk>
UDP echo server: socket(), bind()

Identical to TCP server, except for creating a socket of type SOCK_DGRAM.

```c
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if (sockfd < 0)
    error("ERROR opening socket");

optval = 1;
setsockopt(sockfd, SOL_SOCKET, SO_REUSEADDR,
    (const void *)&optval, sizeof(int));

memset((char *) &serveraddr, 0, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
serveraddr.sin_port = htons((unsigned short)portno);

if (bind(sockfd, (struct sockaddr *) &serveraddr,
    sizeof(serveraddr)) < 0)
    error("ERROR on binding");
```
int main(int argc, char **argv) {
    /* create and configure the UDP socket */

    while(1) {
        /* recvfrom(): read a UDP datagram */
        /* sendto(): echo datagram back to the client */
    }
    return 0;
}
UDP server: recvfrom(), sendto()

The main server loop is a simple sequence of receiving and sending datagrams.

```c
clientlen = sizeof(clientaddr);
while (1) {
    memset(buf, 0, BUFSIZE);
    n = recvfrom(sockfd, buf, BUFSIZE, 0,
                 (struct sockaddr *) &clientaddr, &clientlen);
    if (n < 0)
        error("ERROR in recvfrom");

    n = sendto(sockfd, buf, strlen(buf), 0,
               (struct sockaddr *) &clientaddr, clientlen);
    if (n < 0)
        error("ERROR in sendto");
}
```

Much simpler than the TCP server:

- No accept(), no distinction between child and parent sockets.
- However, user must develop logic for lost or misordered datagrams.
UDP client: socket(), gethostbyname()

Identical to TCP client, except for SOCK_DGRAM.

```c
/* socket: create the socket */
sockfd = socket(AF_INET, SOCK_DGRAM, 0);
if (sockfd < 0)
    error("ERROR opening socket");

/* gethostbyname: get the server's DNS entry */
server = gethostbyname(hostname);
if (server == NULL) {
    fprintf(stderr,"ERROR, no such host as %s\n", hostname);
    exit(1);
}

/* build the server's Internet address */
memset((char *) &serveraddr, 0, sizeof(serveraddr));
serveraddr.sin_family = AF_INET;
memcpy((char *)server->h_addr, (char *)&serveraddr.sin_addr.s_addr,
       server->h_length);
serveraddr.sin_port = htons(portno);
```
The client sends a datagram to the server, waits for the echo, and terminates.

```c
/* get a message from the user */
memset(buf, 0, BUFSIZE);
printf("Please enter msg: ");
fgets(buf, BUFSIZE, stdin);

/* send the message to the server */
serverlen = sizeof(serveraddr);
n = sendto(sockfd, buf, strlen(buf), 0, &serveraddr, serverlen);
if (n < 0)
    error("ERROR in sendto");

/* print the server's reply */
n = recvfrom(sockfd, buf, strlen(buf), 0, &serveraddr,&serverlen);
if (n < 0)
    error("ERROR in recvfrom");

printf("Echo from server: %s", buf);
return 0;
```
Multiplexing I/O: select()

How does a server manage multiple file and socket descriptors?

Example: a TCP server that also accepts user commands from stdin.
  • “c”: print the number of connection requests so far.
  • “q”: terminate the server.

Problem:
  • I/O events can occur asynchronously.
  • Input is available on stdin:
    – e.g., user has typed a line and hit return.
  • Connection request is outstanding on parentfd.
  • Blocking on either fgets() or accept() would create an unresponsive server.

Solution:
  • select() system call.
TCP server based on select()

Use select() to detect events without blocking.

```c
/*
 * main loop: wait for connection request or stdin command.
 * If connection request, then echo input line
 * and close connection. If command, then process.
 */
printf("server> ");
fflush(stdout);
notdone = 1;
while (notdone) {
    /*
      * select: check if the user typed something to stdin or
      * if a connection request arrived.
      */
    FD_ZERO(&readfds);          /* initialize the fd set */
    FD_SET(parentfd, &readfds); /* add socket fd */
    FD_SET(0, &readfds);        /* add stdin fd (0) */
    if (select(parentfd+1, &readfds, 0, 0, 0) < 0) {
        error("ERROR in select");
    }
    ...}
```
TCP server based on select()

First we check for a pending event on stdin.

```c
/* if the user has typed a command, process it */
if (FD_ISSET(0, &readfds)) {
    fgets(buf, BUFSIZE, stdin);
    switch (buf[0]) {
        case 'c': /* print the connection count */
            printf("Received %d conn. requests so far.\n", connectcnt);
            printf("server> ");
            fflush(stdout);
            break;
        case 'q': /* terminate the server */
            notdone = 0;
            break;
        default: /* bad input */
            printf("ERROR: unknown command\n");
            printf("server> ");
            fflush(stdout);
    }
}
```
TCP server based on select()

Next we check for a pending connection request.

```c
/* if a connection request has arrived, process it */
if (FD_ISSET(parentfd, &readfds)) {
    childfd = accept(parentfd, (struct sockaddr *) &clientaddr,
                     &clientlen);
    if (childfd < 0)
        error("ERROR on accept");
    connectcnt++;

    memset(buf, 0, BUFSIZE);
    n = read(childfd, buf, BUFSIZE);
    if (n < 0)
        error("ERROR reading from socket");

    n = write(childfd, buf, strlen(buf));
    if (n < 0)
        error("ERROR writing to socket");
    close(childfd);
}
```

Or, if not just echo server, we can add the childfd to the readfds => we can now receive further messages from this client in the same select:

```c
FD_SET(childfd, &readfds);
// keep track of the max
if (childfd > parentfd+1) {
    fdmax = childfd;
}
```