CpSc 360: Distributed and Network Programming

Socket Programming

James Wang

Read textbook chapter 2, 3, 4, 5.

Important Link

- Source code for sample programs can be found in

  http://www.unpbook.com
- Please download the gzip file and unpack it into your home directory.
- We will use the sample code throughout the entire course.

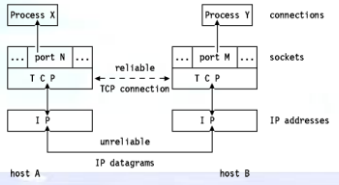
2

TCP Summary

- TCP provides a connection oriented, reliable, byte stream service.
  - The term connection-oriented means the two applications using TCP must establish a TCP connection with each other before they can exchange data.
- It is a full duplex protocol, meaning that each TCP connection supports a pair of byte streams, one flowing in each direction.
- TCP includes a flow-control mechanism for each of these byte streams that allows the receiver to limit how much data the sender can transmit.
- TCP also implements a congestion-control mechanism.

3

TCP connection



- Two processes communicating via TCP sockets. Each side of a TCP connection has a socket which can be identified by the pair *< IP_address, port_number >.*
- Two processes communicating over TCP form a logical connection that is uniquely identifiable by the two sockets involved, that is by the combination *< local_IP_address, local_port, remote_IP_address, remote_port>.*

4

TCP Features

- **Stream Data Transfer:**
  - From the application's viewpoint, TCP transfers a contiguous stream of bytes. TCP does this by grouping the bytes in TCP segments, which are passed to IP for transmission to the destination. TCP itself decides how to segment the data and it may forward the data at its own convenience.
- **Reliability:**
  - TCP assigns a sequence number to each byte transmitted, and expects a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. The receiving TCP uses the sequence numbers to rearrange the segments when they arrive out of order, and to eliminate duplicate segments.
- **Flow Control:**
  - The receiving TCP, when sending an ACK back to the sender, also indicates to the sender the number of bytes it can receive beyond the last received TCP segment, without causing overrun and overflow in its internal buffers. This is sent in the ACK in the form of the highest sequence number it can receive without problems.

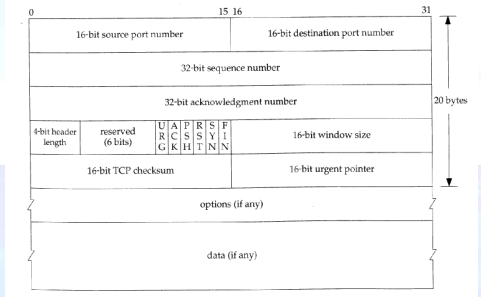
5

TCP Features (Cont.)

- **Multiplexing:**
  - To allow for many processes within a single host to use TCP communication facilities simultaneously, the TCP provides a set of addresses or ports within each host. Concatenated with the network and host addresses from the internet communication layer, this forms a socket. A pair of sockets uniquely identifies each connection.
- **Logical Connections:**
  - The reliability and flow control mechanisms described above require that TCP initializes and maintains certain status information for each data stream. The combination of this status, including sockets, sequence numbers and window sizes, is called a logical connection. Each connection is uniquely identified by the pair of sockets used by the sending and receiving processes.
- **Full Duplex:**
  - TCP provides for concurrent data streams in both directions.

6

TCP Header



TCP Segments

- **When a client requests a connection, it sends a "SYN" segment (a special TCP segment) to the server port.**
- **SYN stands for *synchronize*. The SYN message includes the client's ISN.**
- **ISN is Initial Sequence Number.**
- **Every TCP segment includes a *Sequence Number* that refers to the first byte of *data* included in the segment.**
- **Every TCP segment includes a *Request Number* (*Acknowledgement Number*) that indicates the byte number of the next data that is expected to be received.**
  - All bytes up through this number have already been received.

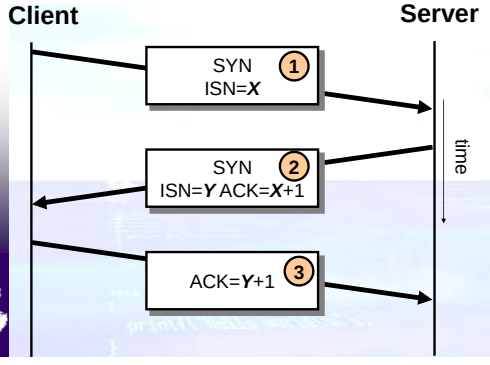
TCP Flags and Options

- **There are a bunch of control flags:**
  - URG: urgent data included.
  - ACK: this segment is (among other things) an acknowledgement.
  - RST: error - abort the session.
  - SYN: synchronize Sequence Numbers (setup)
  - FIN: polite connection termination.
- **MSS: Maximum segment size (A TCP option)**
- **Window: Every ACK includes a Window field that tells the sender how many bytes it can send before the receiver will have to toss it away (due to fixed buffer size).**

TCP Connection Creation

- **A client starts by sending a SYN segment with the following information:**
  - Client's ISN (generated pseudo-randomly)
  - Maximum Receive Window for client.
  - Optionally (but usually) MSS (largest datagram accepted).
  - No payload! (Only TCP headers)
- **When a waiting server sees a new connection request, the server sends back a SYN segment with:**
  - Server's ISN (generated pseudo-randomly)
  - Request Number is Client ISN+1
  - Maximum Receive Window for server.
  - Optionally (but usually) MSS
  - No payload! (Only TCP headers)
- **When the Server's SYN is received, the client sends back an ACK with:**
  - Request Number is Server's ISN+1

TCP 3-way handshake



Question

- **Why do we need 3-way handshake?**
- **HINTS:**
  - TCP is a reliable service.
  - IP delivers each TCP segment.
  - IP is not reliable.
  - Think about 4 messages.

TCP Data and ACK

- Once the connection is established, data can be sent.
- Each data segment includes a sequence number identifying the first byte in the segment.
- Each segment (data or empty) includes a request number indicating what data has been received.

13

Buffering

- The TCP layer doesn't know when the application will ask for any received data.
- TCP buffers incoming data so it's ready when we ask for it.
- Both the client and server allocate buffers to hold incoming and outgoing data
  - The TCP layer does this.
- Both the client and server announce with every ACK how much buffer space remains (the Window field in a TCP segment).

14

Send Buffers

- The application gives the TCP layer some data to send.
- The data is put in a send buffer, where it stays until the data is ACK'd.
  - it has to stay, as it might need to be sent again!
- The TCP layer won't accept data from the application unless (or until) there is buffer space.

15

ACKs

- A receiver doesn't have to ACK every segment (it can ACK many segments with a single ACK segment).
- Each ACK can also contain outgoing data (piggybacking).
- If a sender doesn't get an ACK after some time limit (MSL) it resends the data.

16

TCP Segment Order

- Most TCP implementations will accept out-of-order segments (if there is room in the buffer).
- Once the missing segments arrive, a single ACK can be sent for the whole thing.
- Remember: IP delivers TCP segments, and IP in not reliable - IP datagrams can be lost or arrive out of order.

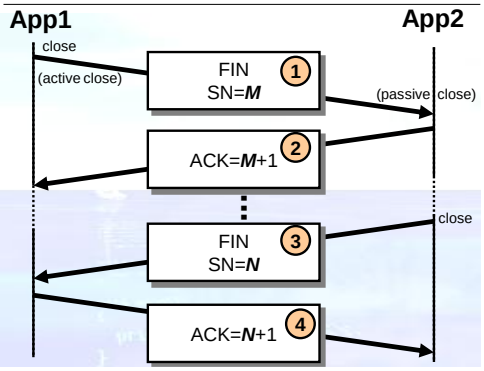
17

Termination

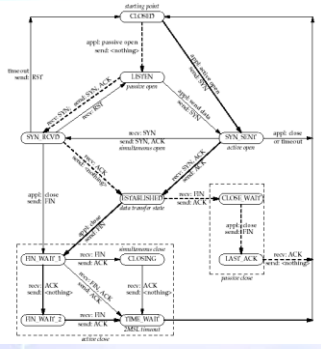
- The TCP layer can send a RST segment that terminates a connection if something is wrong.
- Usually the application tells TCP to terminate the connection politely with a FIN segment.
- Either end of the connection can initiate termination.
- When a FIN is sent, it means the application is done sending data.
- After the FIN is ACK'd by the other end, that end must now send a FIN. That FIN must be ACK'd by this end.

18

TCP Termination



TCP State Transition Diagram

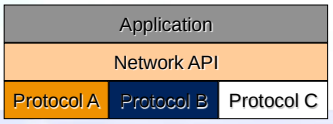


TCP TIME_WAIT State

- **Once a TCP connection has been terminated (the last ACK sent) there is some unfinished business:**
  - What if the ACK is lost? The last FIN will be resent and it must be ACK'd.
  - What if there are lost or duplicated segments that finally reach the destination after a long delay?
- **TCP hangs out for a while to handle these situations.**
- **The duration that this endpoint remains in this state is twice the *maximum segment lifetime* (MSL).**

Network Programming API

- **Network API is normally provided by operating systems.**
- **It provides the interface between application and protocol software.**



- **Network API need:**
  - Generic Programming Interface.
  - Support for message oriented and connection oriented communication.
  - Work with existing I/O services (when this makes sense).
  - Operating System independence.

TCP/IP APIs

- **Sockets:** The Berkeley Unix mechansim for creating a virtual connection between processes. Sockets interface Unix's standard I/O with its network communication facilities.
- **TLI, XTI:** TLI is A protocol-independent interface for accessing network facilities, modeled after the ISO transport layer (level 4), that first appeared in Unix SVR3. XTI (X/OPEN Transport Interface) evolved from TLI, and supports the TLI API for compatibility, with some variations on semantics.
- **Winsock:** A specification for Microsoft Windows network software, describing how applications can access network services, especially TCP/IP.
- **MacTCP:** Part of earlier versions of MacOS that provided access to TCP/IP services. Apple removed MacTCP from MacOS in revision 7.5.3 in favor of the new OpenTransport (OT) TCP/IP stack.

Functions needed for APIs:

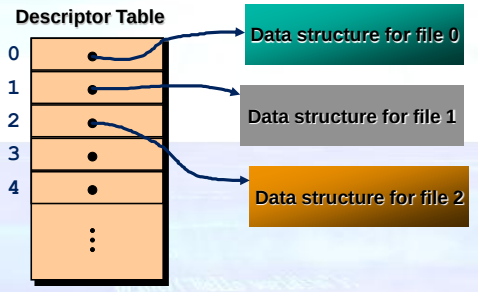
- **Specify local and remote communication endpoints**
- **Initiate a connection**
- **Wait for incoming connection**
- **Send and receive data**
- **Terminate a connection gracefully**
- **Error handling**

Sockets

- **Generic:**
  - support for multiple protocol families.
  - address representation independence
- **Uses existing I/O programming interface as much as possible.**
- **A socket is an abstract representation of a communication endpoint.**
- **Sockets work with Unix I/O services just like files, pipes & FIFOs.**
- **Sockets (obviously) have special needs:**
  - establishing a connection
  - specifying communication endpoint addresses

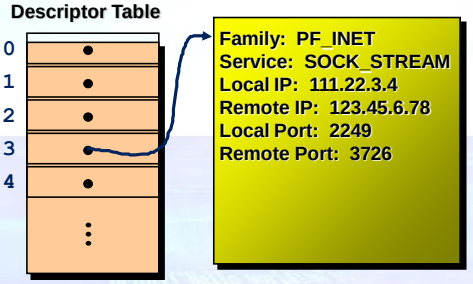
25

Unix Descriptor Table



26

Socket Descriptor Data Structure



27

Creating a Socket

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

- **`family` specifies the protocol family (`PF_INET` for TCP/IP).**
- **`type` specifies the type of service (`SOCK_STREAM`, `SOCK_DGRAM`).**
- **`protocol` specifies the specific protocol (usually 0, which means *the default*).**

28

`socket()`

- **The `socket()` system call returns a socket descriptor (small integer) or -1 on error.**
- **`socket()` allocates resources needed for a communication endpoint - but it does not deal with endpoint addressing.**
- **Specifying an endpoint address.**
  - Remember that the sockets API is generic.
  - There must be a generic way to specify endpoint addresses.
  - TCP/IP requires an IP address and a port number for each endpoint address.
  - Other protocol suites (families) may use other schemes.

29

Specifying an Endpoint Address

- **Remember that the sockets API is generic.**
- **There must be a generic way to specify endpoint addresses.**
- **TCP/IP requires an IP address and a port number for each endpoint address.**
- **Other protocol suites (families) may use other schemes.**

30

POSIX data types

`int8_t`	signed 8bit int
`uint8_t`	unsigned 8 bit int
`int16_t`	signed 16 bit int
`uint16_t`	unsigned 16 bit int
`int32_t`	signed 32 bit int
`uint32_t`	unsigned 32 bit int

`u_char, u_short, u_int, u_long`

`sa_family_t`	address family
`socklen_t`	length of struct
`in_addr_t`	IPv4 address
`in_port_t`	IP port number

Generic socket addresses

```
struct sockaddr {
    uint8_t     sa_len;
    sa_family_t sa_family;
    char        sa_data[14];
};
```

Used by kernel

- `sa_family` specifies the address type.
- `sa_data` specifies the address value.
- Originally sa_len was not there
- Was sa_data size 14 to make the total size 16?
- Depending on the address family, sa_data could be a file name or a socket endpoint...

struct sockaddr_in (IPv4)

```
struct in_addr {
    in_addr_t       s_addr;
};
```

- `in_addr` just provides a name for the 'C' type associated with IP addresses.

```
struct sockaddr_in {
    uint8_t         sin_len;
    sa_family_t     sin_family;
    in_port_t       sin_port;
    struct in_addr  sin_addr;
    char            sin_zero[8];
};
```

- sockaddr_in is a "specialized" sockaddr
- sin_addr could be u_long
- sin_addr is 4 bytes and 8 bytes are unused
- sockaddr_in is used to specify an endpoint

Network Byte Order

- **All values stored in a `sockaddr_in` must be in network byte order.**
  - `sin_port` a TCP/IP port number.
  - `sin_addr` an IP address.
- **Functions:**

```
uint16_t htons(uint16_t);
uint16_t ntohs(uint_16_t);

uint32_t htonl(uint32_t);
uint32_t ntohl(uint32_t);
```

'h' : host byte order 'n' : network byte order

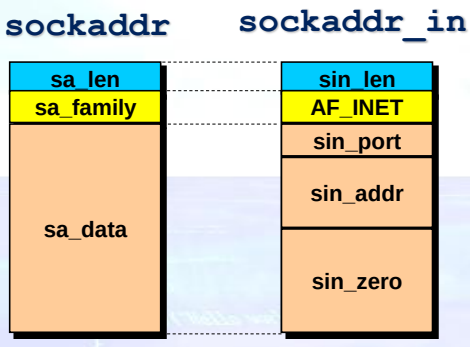
's' : short (16bit) 'l' : long (32bit)

TCP/IP Addresses

- **We don't need to deal with `sockaddr` structures since we will only deal with a real protocol family.**
- **We can use `sockaddr_in` structures.**

BUT: The C functions that make up the sockets API expect structures of type `sockaddr`.

Sockaddr vs. sockaddr_in



Assigning an address to a socket

- **The `bind()` system call is used to assign an address to an existing socket.**

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

const!

- `bind` **returns 0 if successful or -1 on error.**

37

`bind()`

- **calling `bind()` assigns the address specified by the `sockaddr` structure to the socket descriptor.**
- **You can give `bind()` a `sockaddr_in` structure:**

```
bind( mysock, (struct sockaddr*) &myaddr,
      sizeof(myaddr) );
```

- **There are a number of uses for `bind()`:**
  - Server would like to bind to a well known address (port number).
  - Client can bind to a specific port.
  - Client can ask the O.S. to assign *any available* port number.

38

`bind()` Example

```
int mysock,err;
struct sockaddr_in myaddr;

  mysock = socket(PF_INET,SOCK_STREAM,0);
  myaddr.sin_family = AF_INET;
  myaddr.sin_port = htons( portnum );
  myaddr.sin_addr.s_addr = htonl( ipaddress);

  err=bind(mysock, (sockaddr *) &myaddr,
      sizeof(myaddr));
```

39

What is my IP address and port number?

- **IP address:**
  - How can you find out your IP address so that you can tell **`bind()`** ?
  - There is no realistic way for you to know the right IP address to give bind() - what if the computer has multiple network interfaces?
  - specify the IP address as: **`INADDR_ANY`**, this tells the OS to take care of things.

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

- **Port number:**
  - Clients typically don't care what port they are assigned.
  - When you call bind you can tell it to assign you any available port:

```
myaddr.port = htons(0);
```

40

IPv4 Address Conversion

```
int inet_aton( char *, struct in_addr *);
```

Convert ASCII dotted-decimal IP address to network byte order 32 bit value. Returns 1 on success, 0 on failure.

```
char *inet_ntoa(struct in_addr);
```

Convert network byte ordered value to ASCII dotted-decimal (a string).

41

More Socket System Calls

- **General Use:**
  - read()
  - write()
  - close()
- **Connection-oriented (TCP):**
  - connect()
  - listen()
  - accept()
- **Connectionless (UDP):**
  - send()
  - recv()

42

TCP Sockets Programming

- Creating a *passive mode* (server) socket.
- Establishing an application-level *connection*.
- send/receive data.
- Terminating a connection.

43

Creating a TCP socket

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


int sock;

sock = socket(PF_INET,
         SOCK_STREAM,0);

if (sock<0) { /* ERROR */ }
```

44

Binding to well known address

```
int mysock;
struct sockaddr_in myaddr;


   mysock = socket(PF_INET,SOCK_STREAM,0);
   myaddr.sin_family = AF_INET;
   myaddr.sin_port = htons( 80 );
   myaddr.sin_addr.s_addr = htonl( INADDR_ANY );


   bind(mysock, (sockaddr *) &myaddr,
       sizeof(myaddr));
```

45

Establishing a passive mode TCP socket

Passive mode:

- Address already determined.
- Tell the kernel to accept incoming connection requests directed at the socket address.
  - *3-way handshake*
- Tell the kernel to queue incoming connections for us.

46

`listen()`

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

`sockfd` **is the TCP socket (already bound to an address)**

`backlog` **is the number of incoming connections the kernel should be able to keep track of (queue for us).**

`listen()` **returns -1 on error (otherwise 0).**

47

Accepting an incoming connection.

- **Once we call `listen()`, the O.S. will queue incoming connections**
  - Handles the 3-way handshake
  - Queues up multiple connections.
- **When our application is ready to handle a new connection, we need to ask the O.S. for the next connection.**

48

accept()

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

`sockfd` **is the passive mode TCP socket.**

`cliaddr` **is a pointer to *allocated* space.**

`addrlen` **is a *value-result* argument**

- must be set to the size of **`cliaddr`**
- on return, will be set to be the number of used bytes in **`cliaddr`**.

accept() return value

`accept()` **returns a new socket descriptor (small positive integer) or -1 on error.**

After `accept()` **returns a new socket descriptor, I/O can be done using the** `read()` **and** `write()` **system calls.**

`read()` **and** `write()` **operate a little differently on sockets (vs. file operation)!**

Terminating a TCP connection

- **Either end of the connection can call the** `close()` **system call.**
- **If the other end has closed the connection, and there is no buffered data, reading from a TCP socket returns 0 to indicate EOF.**

Client Code

- **TCP clients can call** `connect()` **which:**
  - takes care of establishing an endpoint address for the client socket.
    - Don't need to call bind first, the O.S. will take care of assigning the local endpoint address (TCP port number, IP address).
  - Attempts to establish a connection to the specified server.
    - ***3-way handshake***

connect()

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

`sockfd` **is an already created TCP socket.**

`server` **contains the address of the server (IP Address and TCP port number)**

`connect()` **returns 0 if OK, -1 on error**

Read/Write a TCP socket

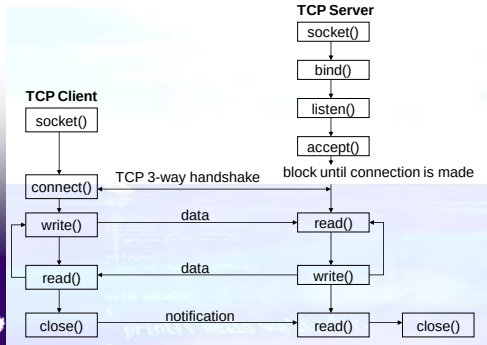
```
int read( int fd, char *buf, int max);
```

- By default **`read()`** will block until data is available.
- reading from a TCP socket may return less than max bytes (whatever is available).

```
int write( int fd, char *buf, int num);
```

- write might not be able to write all num bytes (on a nonblocking socket).
- The book includes readn(), writen() and readline() function definitions.

TCP Sockets Process Relationship



Socket Client Programming

- Create a socket with the socket() system call
- Connect the socket to the address of the server using the connect() system call
- Send and receive data. There are a number of ways to do this, but the simplest is to use the read() and write() system calls.

Client Program Example

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg)
{
    perror(msg);
    exit(0);
}

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;

    char buffer[256];
    if (argc < 3) {
       fprintf(stderr,
         "usage %s hostname port\n", argv[0]);
       exit(0);
    }
    portno = atoi(argv[2]);
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    if (sockfd < 0)
        error("ERROR opening socket");
    server = gethostbyname(argv[1]);
```

```
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);
    if (connect(sockfd,
         &serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");
    printf("Please enter the message: ");
    bzero(buffer,256);
    fgets(buffer,255,stdin);
    n = write(sockfd,buffer,strlen(buffer));
    if (n < 0)
        error("ERROR writing to socket");
    bzero(buffer,256);
    n = read(sockfd,buffer,255);
    if (n < 0)
        error("ERROR reading from socket");
    printf("%s\n",buffer);
    return 0;
}
```


Socket Server Programming

- Create a socket with the socket() system call
- Bind the socket to an address using the bind() system call. For a server socket on the Internet, an address consists of a port number on the host machine.
- Listen for connections with the listen() system call
- Accept a connection with the accept() system call. This call typically blocks until a client connects with the server.
- Send and receive data

Server Program Example

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

void error(char *msg)
{
    perror(msg);
    exit(1);
}

int main(int argc, char *argv[])
{
    int sockfd, newsockfd, portno, clilen;
    char buffer[256];
    struct sockaddr_in serv_addr, cli_addr;
    int n;
    if (argc < 2) {
        fprintf(stderr,
            "ERROR, no port provided\n");
        exit(1);
    }
    sockfd = socket(AF_INET,
                    SOCK_STREAM, 0);
```

```
    if (sockfd < 0)
        error("ERROR opening socket");
    bzero((char *) &serv_addr, sizeof(serv_addr));
    portno = atoi(argv[1]);
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(portno);
    if (bind(sockfd,
          (struct sockaddr *) &serv_addr,
          sizeof(serv_addr)) < 0)
          error("ERROR on binding");
    listen(sockfd,5);
    clilen = sizeof(cli_addr);
    newsockfd = accept(sockfd,
            (struct sockaddr *) &cli_addr,
            &clilen);
    if (newsockfd < 0)
        error("ERROR on accept");
    bzero(buffer,256);
    n = read(newsockfd,buffer,255);
    if (n < 0) error("ERROR reading from socket");
    printf("Here is the message: %s\n",buffer);
    n = write(newsockfd,"I got your message",18);
    if (n < 0) error("ERROR writing to socket");
    return 0;
}
```