TCP

- TCP provides the end-to-end reliable connection that IP alone cannot support

The TCP protocol
- Segment format
- Connection Creation
- Flow control
- Congestion control
- Connection termination

TCP Segment Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>source port</td>
<td>16</td>
</tr>
<tr>
<td>destination port</td>
<td>16</td>
</tr>
<tr>
<td>sequence number</td>
<td>32</td>
</tr>
<tr>
<td>acknowledgment number</td>
<td>32</td>
</tr>
<tr>
<td>window size</td>
<td>16</td>
</tr>
<tr>
<td>urgent pointer</td>
<td>16</td>
</tr>
<tr>
<td>header length</td>
<td>8</td>
</tr>
<tr>
<td>TCP checksum</td>
<td>16</td>
</tr>
<tr>
<td>option (if any)</td>
<td>16</td>
</tr>
<tr>
<td>data (if any)</td>
<td></td>
</tr>
</tbody>
</table>

TCP Connection Establishment
- Three-way handshake

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 ACK segment includes a request number indicating what data has been received. (bytes instead of packets)

Important Information in TCP/IP packet headers
TCP Flow Control

Sender
Application does a 2K write
Application does a 3K write
Sender is blocked
Sender may send up to 2k

Receiver
0 4K empty
Application reads 2K
2K Full
Application reads 2K
1K 2K

TCP Termination

App1
“I have no more data for you.”
FIN_WAIT_1
FIN_WAIT_2
“Over and Out, Goodbye”
TIME_WAIT

App2
FIN
SN=X
ACK=X+1
FIN
SN=Y
ACK=Y+1
CLOSED

FIN

- Either end of the connection can initiate termination.
- A FIN is sent, which means the application is done sending data.
- The FIN is ACK’d.
- The other end must now send a FIN.
- That FIN must be ACK’d.

TCP Sockets Programming

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

Client-Server Communication (TCP)

TCP Server
socket() bind() listen() accept() read() write() close()

TCP Client
socket() connect() read() write() close()

Creating a TCP socket

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

int sock;
sock = socket( AF_INET,
              SOCK_STREAM,
              0);
if (sock<0) { /* ERROR */ }
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

```c
int mysock;
struct sockaddr_in servaddr;
sockfd = socket(AF_INET, SOCK_STREAM, 0);
servaddr.sin_family = AF_INET;
servaddr.sin_port = htons(SERVER_PORT);
servaddr.sin_addr = htonl(SEVER_IP);
if (connect(sockfd, (struct sockaddr *) &servaddr, sizeof(servaddr)) < 0) {
perror("connect error");
exit(1);
}
```

Reading from a TCP socket

```c
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).
- You must be prepared to read data 1 byte at a time!

Writing to a TCP socket

```c
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.

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.

```
#include <unistd.h>
close(int sockfd);
close() returns 0 if OK, -1 on error
```

Server code: 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.

```
#include <sys/socket.h>
#include <netinet/in.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
sockfd is an already created TCP socket.
servaddr contains the address of the server (IP Address and TCP port number)
connect() returns 0 if OK, -1 on error.
```
Binding to well known address

```c
int mysock;
struct sockaddr_in myaddr;

mysock = socket(AF_INET, SOCK_STREAM, 0);
myaddr.sin_family = AF_INET;
myaddr.sin_port = htons(80);
myaddr.sin_addr = htonl(INADDR_ANY);
bind(mysock, (struct sockaddr *) &myaddr, sizeof(myaddr));
```

#include <sys/socket.h>
```c
int bind(int sockfd, const struct sockaddr * myaddr, socklen_t addrlen);
```

bind() return 0 if OK, -1 on error

getsockname()

```c
struct sockaddr_in ss;
socklen_t len;

myaddr.sin_port = htons(0);
myaddr.sin_addr = htonl(INADDR_ANY);
bind(mysock, (struct sockaddr *) &myaddr, sizeof(myaddr));
getsockname(mysock, &ss, &len)
```

#include <sys/socket.h>
```c
int getsockname(int sockfd, struct sockaddr * localaddr, socklen_t * addrlen);
```

bind() return 0 if OK, -1 on error

listen()

```c
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).

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.

accept()

```c
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.

**fork()**

- In Unix the way to create a new process is the `fork()` system call.
- `fork()` is called once but it returns twice

- Return value:
  - 0: return in the child
  - Non-0: the PID of the newly created process

**Client/Server**

- before call to **accept** return

```
129.1.1.200:1500
Client (129.1.1.200)

listenfd=socket(…)
bind(listenfd…)
listen(listenfd,LISTENQ);
For( ; ;) {
    connfd = accept(listenfd, …);
    if ( (pid = fork())==0) {
        close(listenfd);
        doit(connfd);
        close(connfd);
        exit(0);
    }
    close(connfd);
}
```

- After call to **accept** return

```
65.1.1.200
Server (parent)

listenfd=socket(…)
bind(listenfd,LISTENQ);
listen(listenfd,LISTENQ);
fork();
```

```
65.1.1.200
Server (child)
```

```
listenfd=socket(…)
bind(listenfd…)
listen(listenfd,LISTENQ);
fork();
```

```
65.1.1.200
Server (parent)

fork()
```
Client/Server

After call to `accept`

```plaintext
{129.1.1.200:1500, 65.1.1.200:80}  
Client (129.1.1.200)
```

```plaintext
listenfd  
*:80
```

Server (parent)

```plaintext
Server  
(129.1.1.200)
```

```plaintext
connect()

listenfd = socket(...)
bind(listenfd, ...)
listen(listenfd, LISTENQ);
```

```plaintext
For ( ; ; ) {

c = accept(listenfd, ...);
if ( (pid = fork()) ) {

close(listenfd);

doit(c);

close(c);

exit(0);
}
```

```plaintext
close(connfd);
```

```plaintext
Server  
(child)
```

Assignment & Next time

- Reading:
  - UNP 3.9, 4.3-4.9 **

- Next Lecture:
  - RLOGIN
  - TELNET
  - FTP Protocol