Short Connections
vs
Long Connections

Stateful vs Stateless

```
nc
telnet
ssh
Web Browser
```

```
echo       (7)
discard    (9)
daytime    (13)
chargen    (19)
telnet     (23)
time       (37)

FTP      (20/21)
SSH      (22)
DNS       (53)
HTTP     (80/8080)
HTTPS    (443)
DHCP     (546/547)
NTP      (123)
NFS       (2049)
```

Iterative connectionless
Iterative connection-oriented
Concurrent connectionless
Concurrent connection oriented

What is a Socket?

- **A socket is a communication end point to which an application can write data (to be sent to the underlying network) and from which an application can read data. The process/application can be related or unrelated and may be executing on the same or different machines**
- From IPC point of view, a socket is a full-duplex IPC channel that may be used for communication between related or unrelated processes executing on the same or different machines. Both communicating processes need to create a socket to handle their side of communication, therefore, a socket is called an end point of communication
- Available APIs for socket communication are:
- For UNIX: **socket** and **XTI / TLI**
- For Apple Mac: **MacTCP**
- For MS Windows: **Winsock**

Xopen Transport Interface
Transport Layer Interface

Types of Sockets

We will be discussing two types of sockets; the Internet domain sockets and the UNIX domain sockets. Every socket implementation provides at least two types of sockets:

- **TCP/Stream sockets (SOCK_STREAM)**

- **UDP/Datagram sockets (SOCK_DGRAM)**

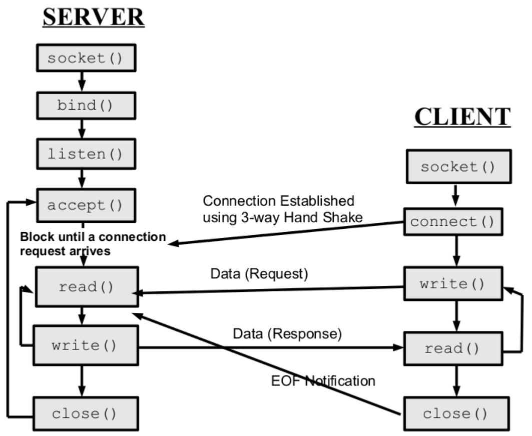
Stream Sockets / TCP Sockets

Stream sockets (SOCK_STREAM) provide a *reliable*, *full-duplex*, *stream-oriented* communication channel. Stream sockets are used to communicate between only two specific processes (point-to-point), and are described as connection-oriented. Do not support broadcasting and multicasting

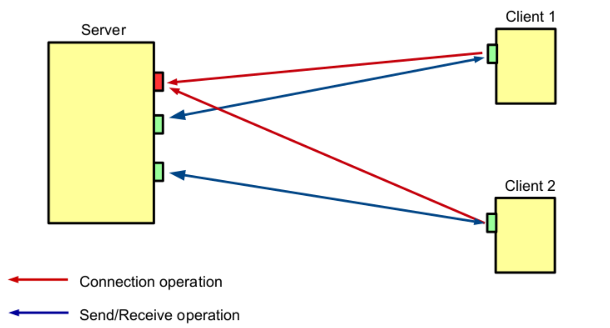
- Full Duplex
- Stream Oriented
- Reliable
- Error detection using checksum
- Flow control using sliding window
- Congestion Control
  - ➔ Sender-side congestion window
  - ➔ Receiver-side advertised window

How Stream Sockets Work?
Behind the curtain

System Call Graph: TCP Sockets



Pictorial Representation of TCP Socket



Pseudocode: TCP Sockets

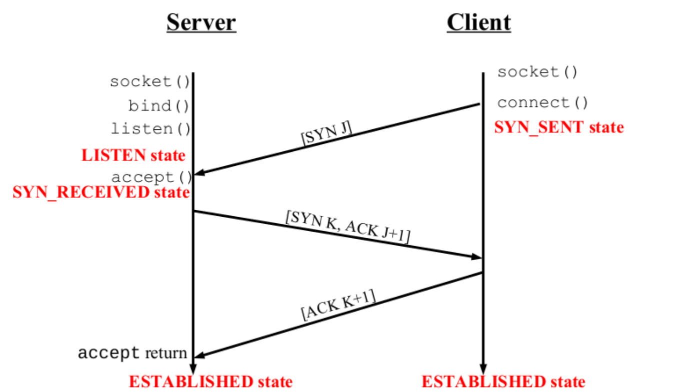
SERVER

```
socket()
bind()
listen()
while(1) {
      accept()
      while(client writes) {
         Read a request
         Perform requested action
         Send a reply
      }
      close client socket
   }
close passive socket
```

CLIENT

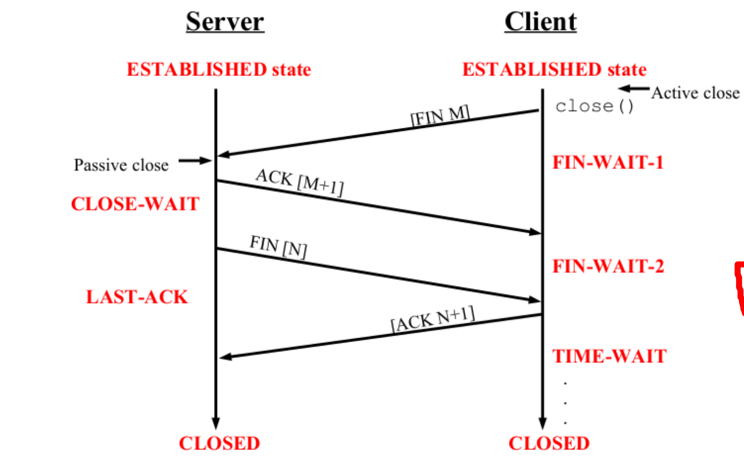
```
socket()
connect()
while(x) {
      write()
      read()
}
close()
```

TCP Three Way Hand Shake



Now reading and writing takes place between client socket and data socket on server side

TCP 4-way Connection Termination



sudo vim /etc/xinet.d/echo edit to yes to no
systemctl stop xinetd.service
systemctl start xinetd.service
netstat -pantu or netsat -pantu | grep 7
check tcp listening in port 7
strace nc localhost 7

for daytime server use port 13

sudo vim /etc/xinet.d/daytime edit yes to no

socket()

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

- **socket()** creates an endpoint for communication
- On success, a file descriptor for the new socket is returned
- On failure, -1 is returned and `errno` is set appropriately
- The first argument `domain` specifies a communication domain under which the communication between a pair of sockets will take place. Communication may only take place between a pair of sockets of the same type
- These families are defined in **/usr/include/x86.../bits/socket.h**

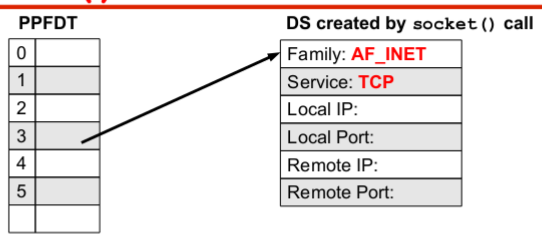
Domain	Comm Performed	Comm between applications	Address format	Address structure
AF_UNIX	Within kernel	On same host	pathname	sockaddr_un
AF_INET	Via IPv4	On hosts connected via an IPv4 network	32-bt IPv4 addr + 16-bit port#	sockaddr_in
AF_INET6	Via IPv6	On hosts connected via IPv6 network	128-bit IPv6 addr + 16-bit port#	sockaddr_in6

socket() . . .

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

- The second argument `type` specifies the communication semantics. These types are defined in the header file `/usr/include/x86.../bits/socket_type.h`. Most common types are SOCK_STREAM and SOCK_DGRAM
- The 3rd argument specifies the protocol to be used within the network code inside the kernel, not the protocol between the client and server. Just set this argument to "0" to have `socket()` choose the correct protocol based on the `type`. You may use constants, like IPPROTO_TCP, IPPROTO_UDP. You may use `getprotobyname()` function to get the official protocol name (discussed later).You may look at `/etc/protocols` file for details
- To view more details about these constants visit following man pages:
  - **$man 7 tcp, udp, raw, unix, ip, socket**
  - **$man 5 protocols**

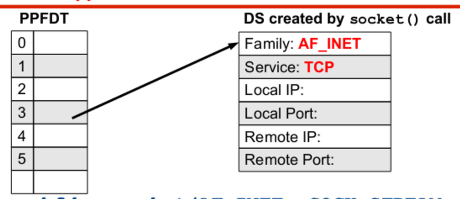
socket() . . .



`int sockfd = socket(AF_INET, SOCK_STREAM, 0);`

- The socket data structure contains several pieces of information for the expected style of IPC, including family/domain, service type, local IP, local port, remote IP, and remote port
- UNIX kernel initializes the first two fields when a socket is created
- When the local address is stored in socket data structure we say that the socket is half associated
- When both local and remote addresses are stored in socket data structure, we say that socket is fully associated

socket() . . .



`int sockfd = socket(AF_INET, SOCK_STREAM, 0);`

How addresses in socket data structure are populated

For Client

- Remote end point address is populated by `connect()`
- Local end point address is automatically populated by TCP/IP s/w when client calls `connect()`

For Server

- Local end point addresses are populated by `bind()`
- Remote end point addresses are populated by `accept()`

connect()

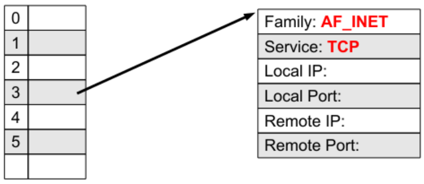
```
int connect(int sockfd, const struct sockaddr *svr_addr,
            int addrlen);
```

- The `connect()` system call connects the socket referred to by the descriptor `sockfd` to the remote server (specified by `svr_addr`)
- If we haven't call `bind()`, (which we normally don't in client), it automatically chooses a local end point address for you
- On success, zero is returned, and the `sockfd` is now a valid file descriptor open for reading and writing. Data written into this file descriptor is sent to the socket at the other end of the connection, and data written into the other end may be read from this file descriptor
- TCP sockets may successfully connect only once. UDP sockets normally do not use `connect()`, however, connected UDP sockets may use `connect()` multiple times to change their association
- When used with `SOCK_DGRAM` type of socket, the `connect()` call simply stores the address of the remote socket in the local socket's data structure, and it may communicate with the other side using `read()` and `write()` instead of using `recvfrom()` and `sendto()` calls

connect()...

`connect()` performs four tasks

- Ensure that the specified `sockfd` is valid and that it has not already been connected
- Fills in the remote end point address in the (client) socket from the second argument
- Automatically chooses a local end point address by calling TCP/IP software
- Initiates a TCP connection (3 way handshake) and returns a value to tell the caller whether the connection succeeded



Socket Address Structures

Generic Socket Address structure: This is a basic template on which other address data structures of different domains are based. When `sa_family` is AF_UNIX the `sa_data` field is supposed to contain a pathname as the socket's address. When `sa_family` is AF_INET the `sa_data` field contains both an IP address and a port number

```
struct sockaddr{
   u_short sa_family;
   char    sa_data[14];
}
```

Internet Socket Address Structure:

```
struct sockaddr_in{
   u_short          sin_family;
   u_short          sin_port;
   struct in_addr   sin_addr;
   char             sin_zero[8];
}
```

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

UNIX Domain Socket Address Structure:

```
struct sockaddr_un{
   short  sun_family;
   char   sun_path;
}
```

Populating Address Structure

- **Example:** We normally need to populate the address structure and then pass it to `connect()`. Following is the code snippet that do the task:

```
struct sockaddr_in svr_addr;
svr_addr.sin_family = AF_INET;
svr_addr.sin_port = htons(54154);
inet_aton("127.0.0.1", &svr_addr.sin_addr);
memset(&(svr_addr.sin_zero), '\0, sizeof(svr_addr.sin_zero));
connect(sockfd,(struct sockaddr*)&svr_addr,sizeof(svr_addr));
```

- **Question:** Why we need to cast the `sockaddr_in` to generic socket address structure `sockaddr`?
- **Answer:** Address structures (of all families) need to be passed to `bind()`, `connect()`, `accept()`, `sendto()`, `recvfrom()`. In 1982, there was no concept of `void*`, so the designers defined a generic socket address structure

Little Endian vs Big Endian

- Byte order is the attribute of a processor that indicates whether integers are represented from left to right or right to left in the memory
- In **Little Endian Byte Order**, the low-order byte of the number is stored in memory at the *lowest address* and the high-order byte of the same number is stored at the highest address
- In **Big Endian Byte Order**, the low-order byte of the number is stored in memory at the *highest address* and the high-order byte of the same number is stored at the lowest address

```
short int var = 0x0001;
char *byte = (char*)&var;
if (byte[0] == 1)
   printf("Little Endian");
else
   printf("Big Endian");
```

Byte Order and ByteOrdering Functions

```
uint16_t htons(uint16_t host16bitvalue);
uint16_t htonl(uint32_t host32bitvalue);
```
Returns: value of arg passed is converted to NBO
```
uint16_t ntohs(uint16_t net16bitvalue);
uint16_t htons(uint32_t net32bitvalue);
```
Returns: value of arg passed is converted to HBO

- The API `htons()` is used to convert a 16-bits data from *host byte order to network byte order* such as TCP or UDP port number
- The API `htonl()` is used to convert a 32-bits data from *host byte order to network byte order* such as IPv4 address
- The API `ntohs()` is used to convert a 16-bits data from *network byte order to host byte order* such as TCP or UDP port number
- The API `ntohl()` is used to convert a 32-bits data from *network byte order to host byte order* such as IPv4 address

Address Format Conversion Functions

```
in_addr_t inet_addr(const char* str);
int inet_aton(const char* str,struct in_addr *addr)
```

- Both of these functions are used to convert the IPv4 internet address from dotted decimal C string format pointed to by `str` to 32-bit binary network byte ordered value
- The `inet_addr()` return this value, while `inet_aton()` function stores it through the pointer `addr`
- The newer function `inet_aton()` works with both IPv4 and IPv6, so one should use this call in the code even if working on IPv4

read() and write()

```
ssize_t read(int fd, void* buf, size_t count);
ssize_t write(int fd, const void* buf, size_t count);
```

- The **read()** and **write()** system calls can be used to read/write from files, devices, sockets, etc. (with any type of sockets stream or datagram)
- The **read()** call **attempts** to read up to `count` bytes from file descriptor `fd` into the buffer starting at `buf`. If no data is available read blocks. On success returns the number of bytes read and on error returns -1 with `errno` set appropriately
- The **write()** call writes `count` number of bytes starting from memory location pointed to by `buf`, to file descriptor `fd`. On success returns the number of bytes actually written and on error returns -1 with errno set appropriately
- The **send()** and **recv()** calls can be used for communicating over stream sockets or connected datagram sockets. If you want to use regular unconnected datagram sockets (UDP), you need to use the **sendto()** and **recvfrom()**

send()

```
int send(int sockfd, const void* buf, int count,int flags);
```

- The **send()** call writes the `count` number of bytes starting from memory location pointed to by `buf`, to file descriptor `sockfd`
- The argument `flags` is normally set to zero, if you want it to be "normal" data. You can set flag as `MSG_OOB` to send your data as "out of band". Its a way to tell the receiving system that this data has a higher priority than the normal data. The receiver will receive the signal `SIGURG` and in the handler it can then receive this data without first receiving all the rest of the normal data in the queue
- The **send()** call returns the number of bytes actually sent out and this might be less than the number you told it to send. If the value returned by **send()** does not match the value in `count`, it's up to you to send the rest of the string
- If the socket has been closed by any side, the process calling **send()** will get a `SIGPIPE` signal

recv()

```
int recv(int sockfd, void* buf, int count, int flags);
```

- The **recv()** call **attempts** to read up to `count` bytes from file descriptor `sockfd` into the buffer starting at `buf`. If no data is available it blocks
- The argument `flags` is normally set to zero, if you want it to be a regular vanilla `recv()`, you can set flag as `MSG_OOB` to receive out of band data. This is how to get data that has been sent to you with the `MSG_OOB` flag in `send()` As the receiving side, you will have had signal `SIGURG` raised telling you there is urgent data. In your handler for that signal, you could call `recv()` with this `MSG_OOB` flag
- The call returns the number of bytes actually read into the buffer, or -1 on error
- If **recv()** returns 0, this can mean only one thing, i.e., remote side has closed the connection on you

Reading in a Loop

- The data from a TCP socket should always be read in a loop until the desired amount of data has been received
- A sample code snippet that do this job is shown:

```
char msg[128];
int n, nread, nremaining;
for(n=0, nread=0; nread < 128; nread += n){
   nremaining = 128 - nread;
   n = read(sockfd, &msg[nread], nremaining);
   if (n == -1) {perror("read failed"); exit(1);}
   }
printf("%s\n",msg);
```

close()

```
int close(int fd);
```

- After a process is done using the socket, it can call `close()` to close it, and it will be freed up, never to be used again by that process
- On success returns zero, or -1 on error and `errno` will be set accordingly
- The remote side can tell if this happens in one of two ways:
  - If the remote side calls `read()`, it will return zero
  - If the remote side calls `write()`, it will receive a signal `SIGPIPE` and `write()` will return -1 and `errno` is set to `EPIPE`
- In practice, Linux implements a reference count mechanism to allow multiple processes to share a socket. If **n** processes share a socket, the reference count will be **n**. `close()` decrements the reference count each time a process calls it. Once the reference count reaches zero (i.e., all processes have called `close()`) the socket will be deallocated