Reading 10

✱ Network Programming:

- A socket is a way to speak to other programs using standard Unix file descriptors.

- 2 main types of internet sockets:
  1. Stream Sockets
     - ↳ reliable, error-free two way connected connection streams.
     - ↳ if you output two items into the socket in the order "1,2", they will arrive in the order "1,2" at the opposite end.
  2. Datagram Sockets
     - ↳ no connection needed
     - ↳ if you send a datagram, it may arrive, it may arrive out of order, if it arrives, then the data within the packet will be error free.

- In IPv4 local host is 127.0.0.1
  - ↳ but in IPv6 local host is ::1

- In IPv6 IP addresses look like this:

hex

```
2001:0db8:c9d2:0012:0000:0000:0000:0051
2001:db8:c9d2:12::51

2001:0db8:ab00:0000:0000:0000:0000:0000
2001:db8:ab00::

0000:0000:0000:0000:0000:0000:0000:0001
::1
```

} equivalent

- Ip addresses have a network portion and a host portion.

- Big-Endian is also called Network Byte Order because that's the order network types like.

- Little-Endian is also called host-byte order.

-

Function	Description
htons()	host to network short
htonl()	host to network long
ntohs()	network to host short
ntohl()	network to host long

↳ you want to convert numbers to Network Byte Order before they go out on the wire
↳ and convert to Host Byte Order as they come in off the wire.

- A socket descriptor is just an int.

- getaddrinfo() : helps set up structs you need later on

```c
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

int getaddrinfo(const char *node,   // e.g. "www.example.com" or IP
                const char *service,  // e.g. "http" or port number
                const struct addrinfo *hints,
                struct addrinfo **res);
```

```c
int status;
struct addrinfo hints;
struct addrinfo *servinfo;  // will point to the results

memset(&hints, 0, sizeof hints); // make sure the struct is empty
hints.ai_family = AF_UNSPEC;     // don't care IPv4 or IPv6
hints.ai_socktype = SOCK_STREAM; // TCP stream sockets
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

if ((status = getaddrinfo(NULL, "3490", &hints, &servinfo)) != 0) {
    fprintf(stderr, "gai error: %s\n", gai_strerror(status));
    exit(1);
}

// servinfo now points to a linked list of 1 or more
// struct addrinfos

// ... do everything until you don't need servinfo anymore ....

freeaddrinfo(servinfo); // free the linked-list
```

- socket() : get the file descriptor

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

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

```c
int s;
struct addrinfo hints, *res;

// do the lookup
// [pretend we already filled out the "hints" struct]
getaddrinfo("www.example.com", "http", &hints, &res);

// again, you should do error-checking on getaddrinfo(), and walk
// the "res" linked list looking for valid entries instead of just
// assuming the first one is good (like many of these examples do).
// See the section on client/server for real examples.

s = socket(res->ai_family, res->ai_socktype, res->ai_protocol);
```

- bind() : associating a socket with a port on your local machine

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

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

```c
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

getaddrinfo(NULL, "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// bind it to the port we passed in to getaddrinfo():

bind(sockfd, res->ai_addr, res->ai_addrlen);
```

- connect() : connecting to a remote host

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

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

```c
struct addrinfo hints, *res;
int sockfd;

// first, load up address structs with getaddrinfo():

memset(&hints, 0, sizeof hints);
hints.ai_family = AF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;

getaddrinfo("www.example.com", "3490", &hints, &res);

// make a socket:

sockfd = socket(res->ai_family, res->ai_socktype, res->ai_protocol);

// connect!

connect(sockfd, res->ai_addr, res->ai_addrlen);
```

- listen() : listening for incoming connections

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

```c
getaddrinfo();
socket();
bind();
listen();
/* accept() goes here */
```

- accept(): Someone is trying to connect() to your machine on a port you are listening() on, you call accept() and you get the pending connection. It'll return a brand new socket file descriptor to use for this single connection.

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

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

```c
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

#define MYPORT "3490"  // the port users will be connecting to
#define BACKLOG 10     // how many pending connections queue holds

int main(void)
{
    struct sockaddr_storage their_addr;
    socklen_t addr_size;
    struct addrinfo hints, *res;
    int sockfd, new_fd;

    // !! don't forget your error checking for these calls !!

    // first, load up address structs with getaddrinfo():

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;  // use IPv4 or IPv6, whichever
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE;     // fill in my IP for me

    getaddrinfo(NULL, MYPORT, &hints, &res);

    // make a socket, bind it, and listen on it:

    sockfd = socket(res->ai_family, res->ai_socktype,
                                                res->ai_protocol);
    bind(sockfd, res->ai_addr, res->ai_addrlen);
    listen(sockfd, BACKLOG);

    // now accept an incoming connection:

    addr_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr,
                                                    &addr_size);

    // ready to communicate on socket descriptor new_fd!
    .
    .
    .
```

- send() and recv() : for communicating over socket streams or connected datagram sockets.

```
int send(int sockfd, const void *msg, int len, int flags);
```

```
1  char *msg = "Beej was here!";
2  int len, bytes_sent;
3  .
4  .
5  .
6  len = strlen(msg);
7  bytes_sent = send(sockfd, msg, len, 0);
8  .
9  .
10 .
```

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

- sendto() and recvfrom() : for datagram sockets that are NOT connected.

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags,
           const struct sockaddr *to, socklen_t tolen);
```

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags,
             struct sockaddr *from, int *fromlen);
```

- close() and shutdown() : closing the connection on your socket descriptor

```
close(sockfd);
```

```
int shutdown(int sockfd, int how);
```

how	Effect
0	Further receives are disallowed
1	Further sends are disallowed
2	Further sends and receives are disallowed (like `close()`)

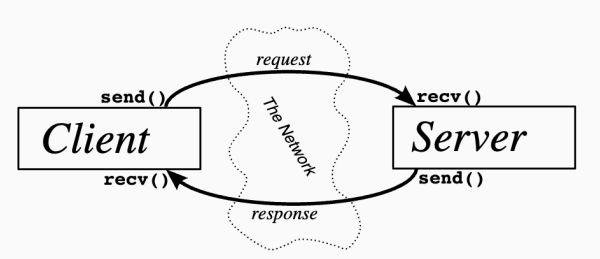
- **getpeername()** : tells you who is at the other end of a connected socket stream

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

int getpeername(int sockfd, struct sockaddr *addr, int *addrlen);
```

- **gethostname()** : returns the name of the computer that your program is running on.

```c
#include <unistd.h>

int gethostname(char *hostname, size_t size);
```

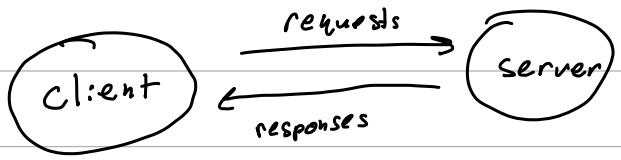
- Client - Server Diagram :



- Use 'netstat' command to get a list of open sockets on the system.

- use 'route' or 'netstat -r' or 'ip route' to view the routing table.

☆ How the Web Works



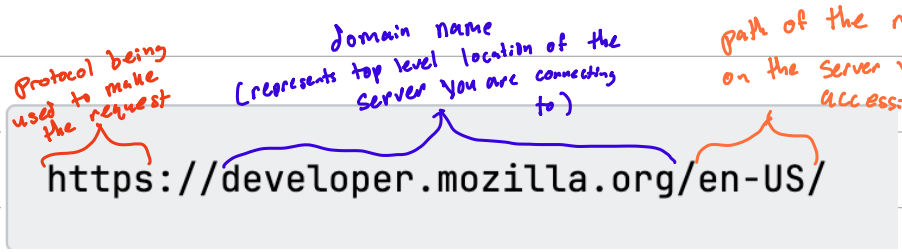
- **Clients** : typical web user's internet connected devices (computer, phone, etc.)
- **Servers** : computers that store webpages, sites, or, apps.

- The client downloads a copy of the webpage and the browser renders it.

- For data to get back and forth you need:
  - internet connection
  - TCP/IP : communication protocols that define how data should travel across the internet.
  - DNS : like an address book for websites
  - HTTP: an application protocol that defines a language for clients and servers to speak to each other.
  - Files: Usually two types
    - ↳ code
    - ↳ assets: images, music, video, etc.

- Packets: format in which data is transferred between the client and server
  - ↳ when data is sent across the web, it is sent in multiple chunks called packets. Each packet contains:
    - A header : which includes details such as server and client IP addresses, the packet number, total # of packets in the transmission, details of protocols.
    - A payload : contains the actual data sent in the packet.

- URL broken down:



☆ What is a URL:

- the address of a unique resource on the internet

- URL Breakdown :

  - Scheme: indicates the protocol
  - Authority:
    - ↳ Domain Name: which web server is being requested
    - ↳ Port: indicates the technical "gate" used to access the resources on the web server.
  - Path to resource: path to the resource on the web server
  - Parameters: key-value pairs so web server can do extra stuff
    - ↳ format: `?key1=value1&key2=value2`
  - Anchor: anchor to another part of the resource itself. "Bookmark"

- There are absolute and relative URL's.
  - ↳ often used within HTML

- URL usernames and passwords (uncommon):

```
https://username:password@www.example.com:80/
```

- You should build semantic URLs

- Data URLs: URLs prefixed with `data:` scheme, allow content creators to embed small files inline in documents.