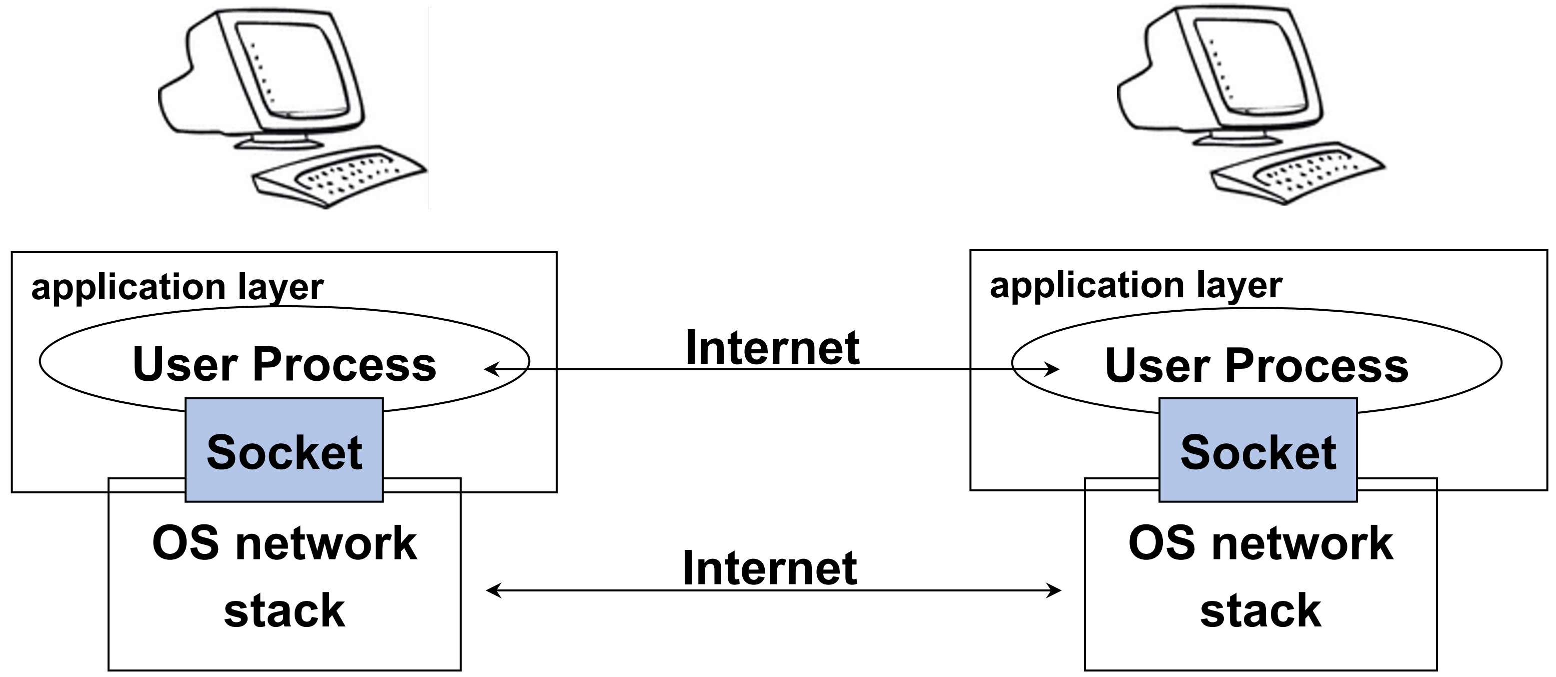
COS461 Precept 1

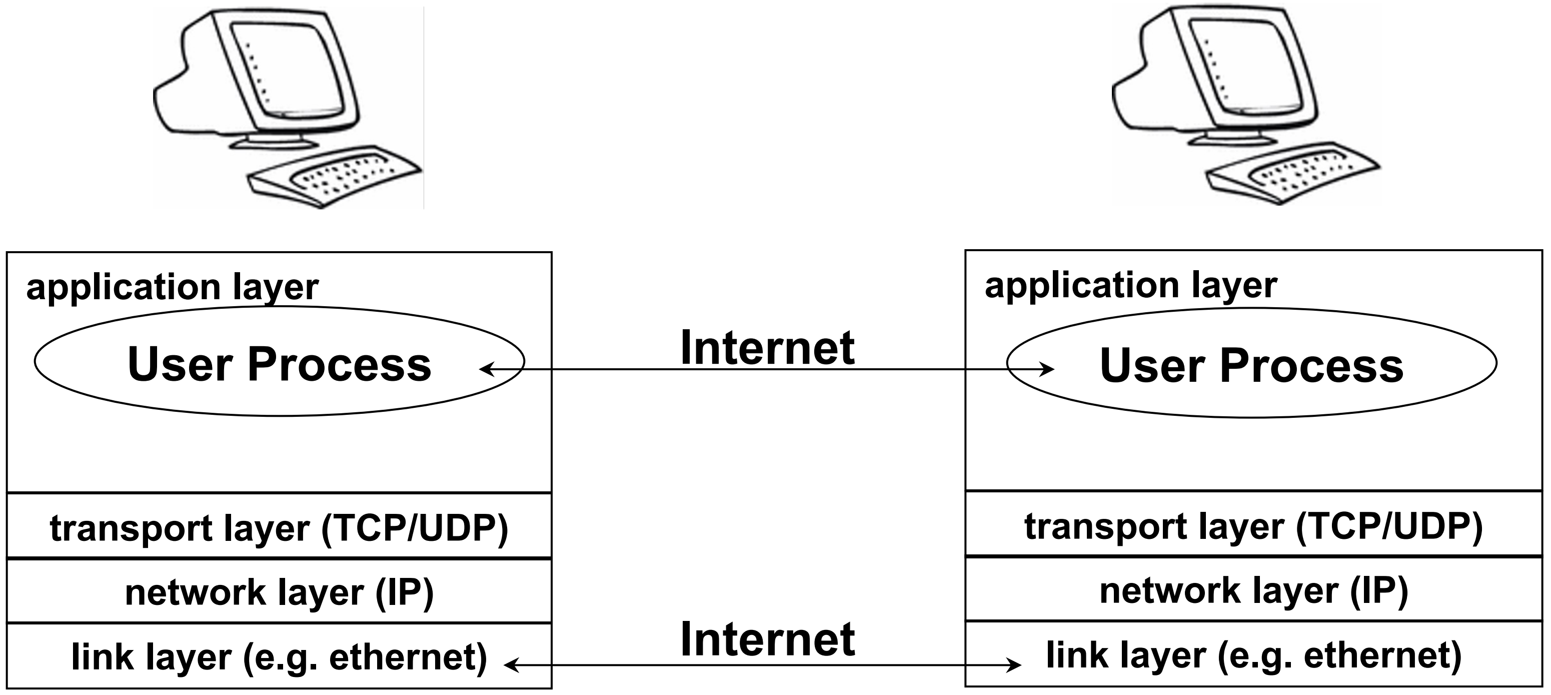
2/3/2022

Socket and Process Communication



The interface that the OS provides to its networking subsystem

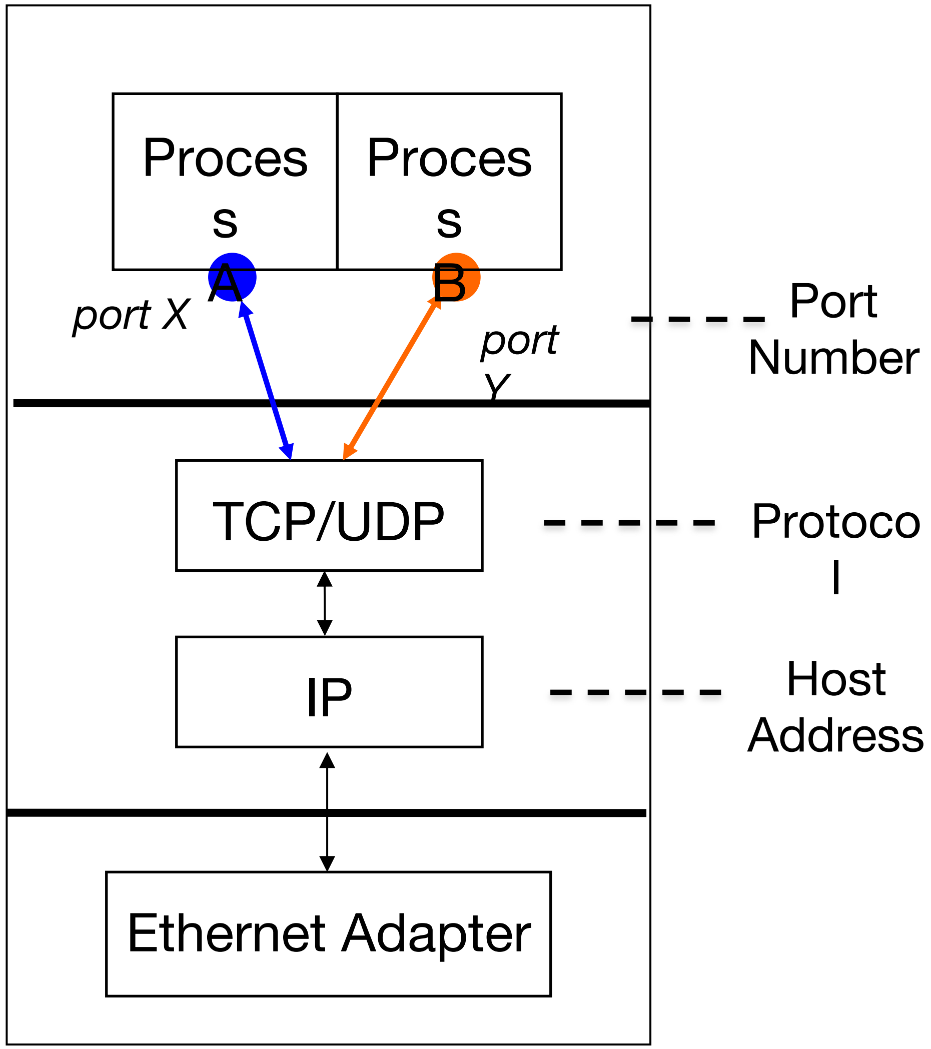
Socket and Process Communication



The interface that the OS provides to its networking subsystem

Socket and Process Communication

- Receiving host
  - Destination **address** that uniquely identifies host
  - **IP address**: 32-bit quantity ("1.2.3.4")
- Receiving socket
  - Host may be running many different processes
  - Destination **port** that uniquely identifies socket
  - **Port number:** 16-bits ("80")



Socket and Process Communication

- Two types of Sockets:
  - Stream Sockets: ‘Reliable’ TCP connections
    - Guarantees that the message sent will be delivered in the same order as they are sent.
    - A connection is established between the two ends before message is sent
  - Datagram Sockets: UDP Connections
    - No guarantee on message delivery – maybe lost, may get reordered.
    - Connectionless – The message is simply sent.

Socket Programming

- The idea of socket programming is to use the socket interface for communicating between processes/applications running across the network.
  - You may want to download a file stored on a server.
  - You may want to create an application which to chat with your friends.

Let us look at socket programming briefly using C. The key ideas will remain same across different languages, only the syntax will differ.

Get the Address of the service/device you are trying to access/want to use : getaddrinfo

```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

// get ready to connect
status = getaddrinfo("www.example.net", "3490", &hints, &servinfo);

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

// etc.
```

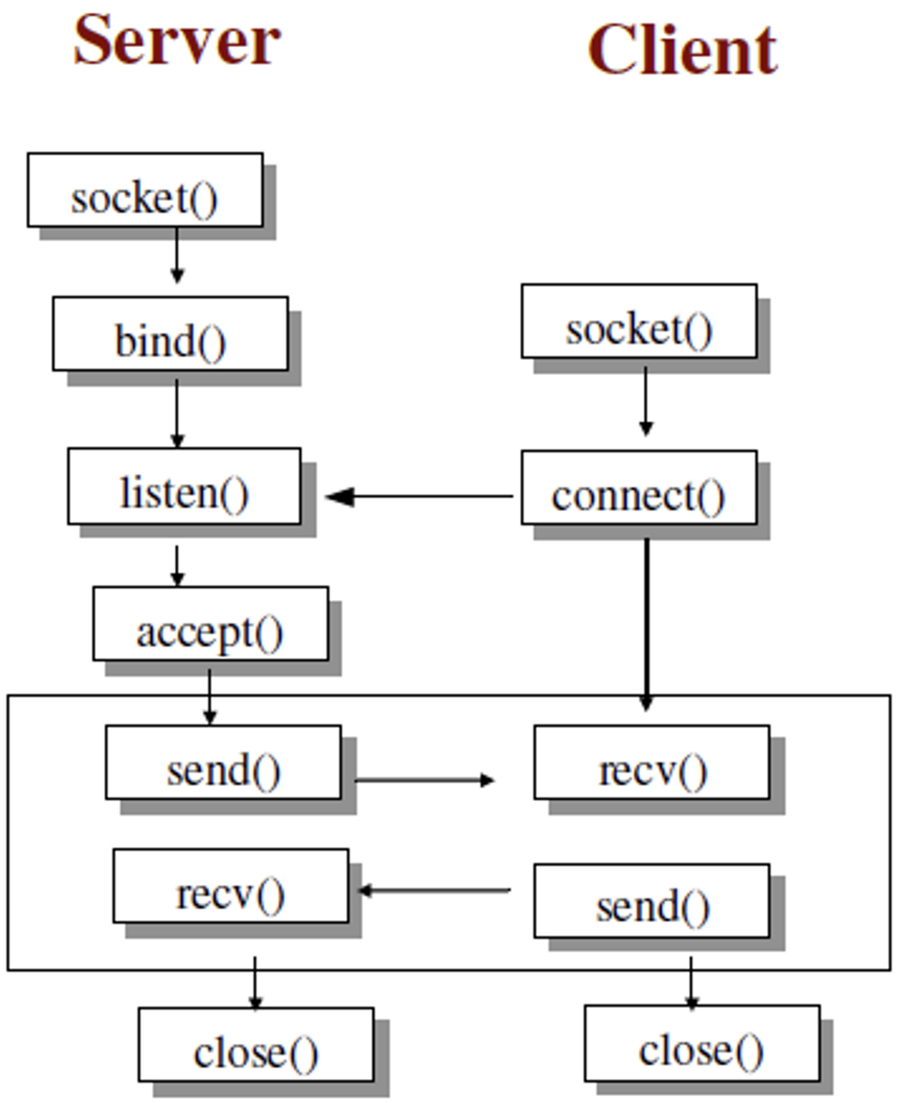
Problem: Create a program that receives messages over the network and act accordingly / create a SERVER

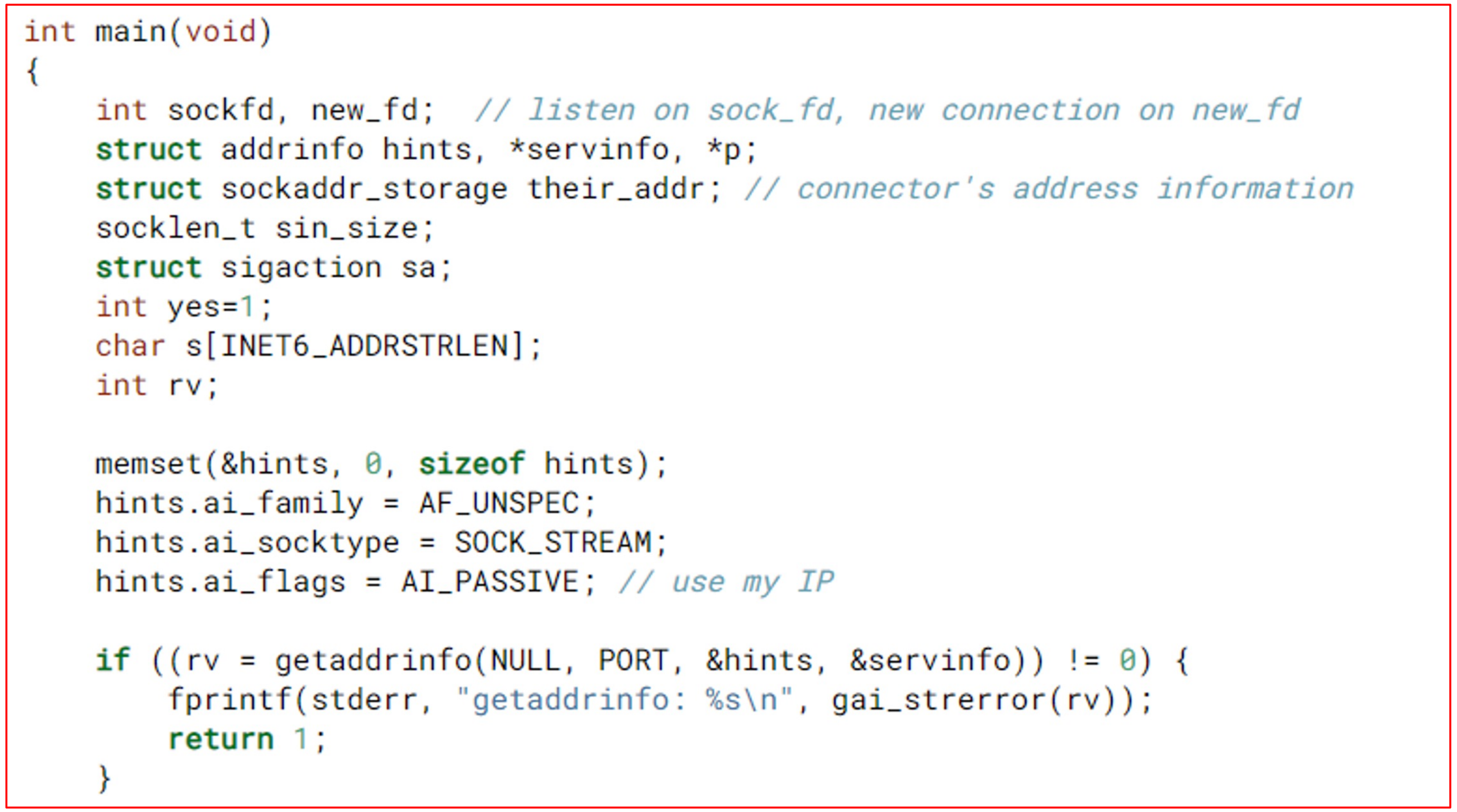
Lets think on what all we need to do :

Goal: To create a socket that can be used to listen to incoming connections.

1. Create a TCP socket.
2. Associate the socket with a PORT number.
3. Listen to incoming connections.
4. Accept incoming connections
5. Send a message to client/receive a message from client

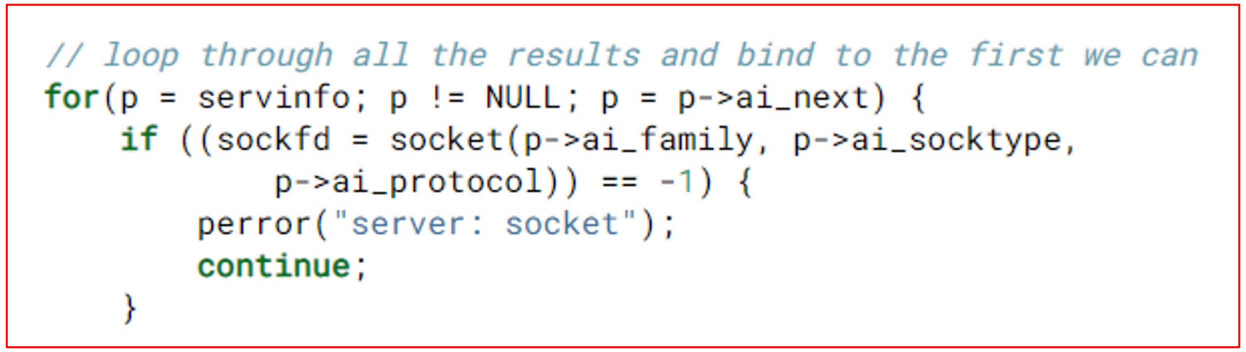
Connection Oriented Protocol





```c
int main(void)
{
    int sockfd, new_fd;  // listen on sock_fd, new connection on new_fd
    struct addrinfo hints, *servinfo, *p;
    struct sockaddr_storage their_addr; // connector's address information
    socklen_t sin_size;
    struct sigaction sa;
    int yes=1;
    char s[INET6_ADDRSTRLEN];
    int rv;

    memset(&hints, 0, sizeof hints);
    hints.ai_family = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags = AI_PASSIVE; // use my IP

    if ((rv = getaddrinfo(NULL, PORT, &hints, &servinfo)) != 0) {
        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
        return 1;
    }
```

Use getaddrinfo to collect information about our own IP.



```c
// loop through all the results and bind to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("server: socket");
        continue;
    }
```

```c
if (bind(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
    close(sockfd);
    perror("server: bind");
    continue;
}
```

```c
if (listen(sockfd, BACKLOG) == -1) {
    perror("listen");
    exit(1);
}
```

```c
while(1) {  // main accept() loop
    sin_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
    if (new_fd == -1) {
        perror("accept");
        continue;
    }
```

```c
if (!fork()) { // this is the child process
    close(sockfd); // child doesn't need the listener
    if (send(new_fd, "Hello, world!", 13, 0) == -1)
        perror("send");
    close(new_fd);
    exit(0);
}
```

Problem: Create a program that connects to another process / create a CLIENT

Lets think on what all we need to do :

Goal: To create a socket that can be used to listen to incoming messages

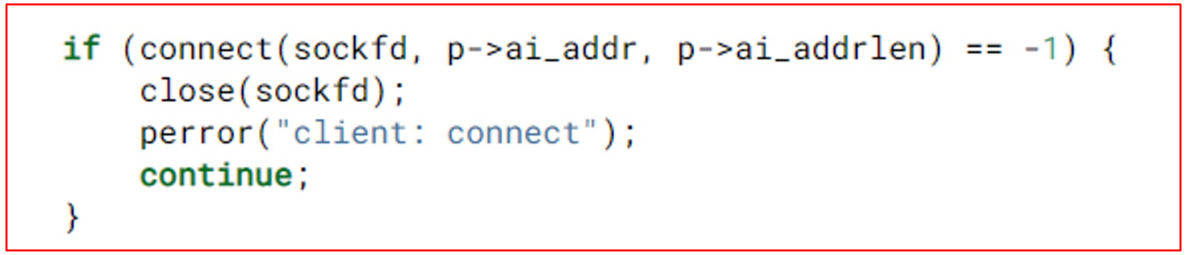
1. Create a TCP socket.
2. ~~Associate the socket with a PORT number.~~
3. Connect to remote process.
4. Send a message to server/receive a message from server.

```c
int sockfd, numbytes;
char buf[MAXDATASIZE];
struct addrinfo hints, *servinfo, *p;
int rv;
char s[INET6_ADDRSTRLEN];

if (argc != 2) {
    fprintf(stderr,"usage: client hostname\n");
    exit(1);
}

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

if ((rv = getaddrinfo(argv[1], PORT, &hints, &servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(rv));
    return 1;
}
```

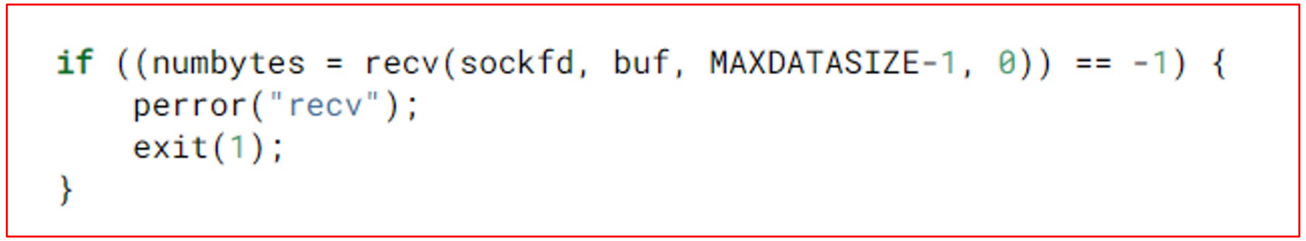
Use getaddrinfo to collect information about server IP.

```c
// loop through all the results and connect to the first we can
for(p = servinfo; p != NULL; p = p->ai_next) {
    if ((sockfd = socket(p->ai_family, p->ai_socktype,
            p->ai_protocol)) == -1) {
        perror("client: socket");
        continue;
    }
```

Bind is not needed as we don’t intend to act as a listener for incoming connections and hence associating a fix port number is not necessary



```
if (connect(sockfd, p->ai_addr, p->ai_addrlen) == -1) {
    close(sockfd);
    perror("client: connect");
    continue;
}
```



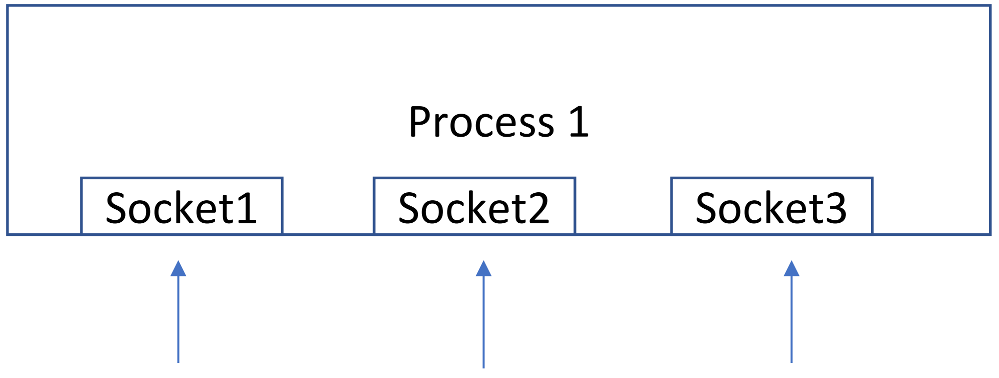
```
if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
    perror("recv");
    exit(1);
}
```

Blocking calls vs Polling

```
if ((numbytes = recv(sockfd, buf, MAXDATASIZE-1, 0)) == -1) {
    perror("recv");
    exit(1);
}
```

```
while(1) {  // main accept() loop
    sin_size = sizeof their_addr;
    new_fd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size);
    if (new_fd == -1) {
        perror("accept");
        continue;
    }
```

accept and recv are both blocking calls, it means that at these calls the program halts and the control is transferred back to the OS. The OS returns to them when a new connection is available or new data is received.

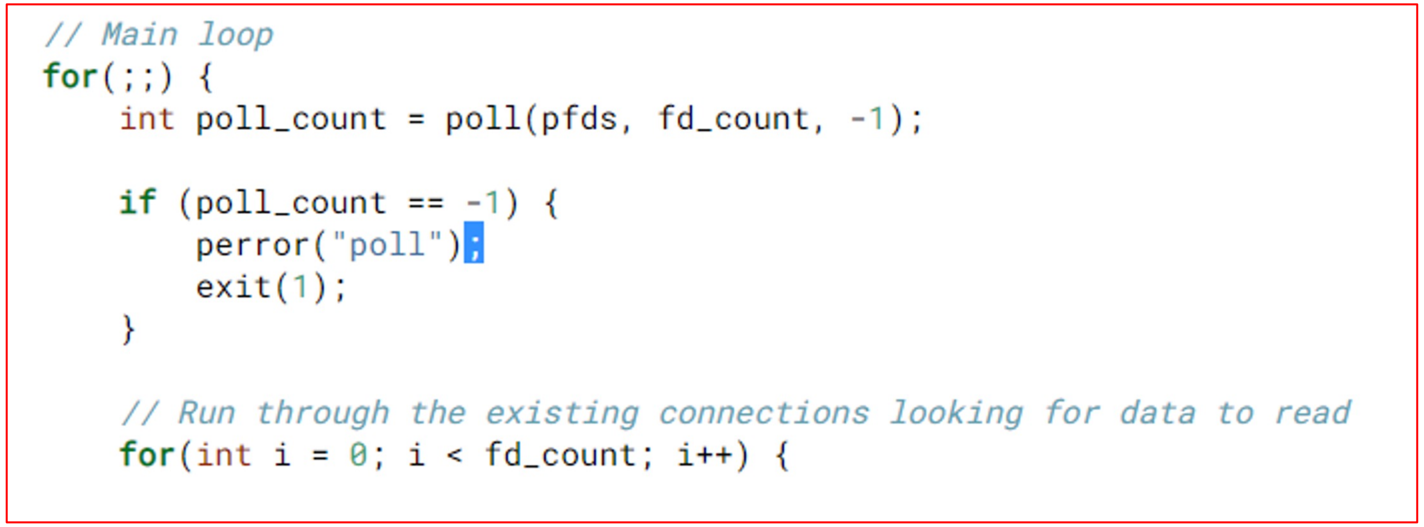


Can receive on only one socket at a time

Solution?

Non-Blocking calls/polling based design

Idea: OS will collect the relevant events (e.g. received messages) in a queue and it is the programs responsibility to repeatedly check the queues and address the events.



```
// Main loop
for(;;) {
    int poll_count = poll(pfds, fd_count, -1);

    if (poll_count == -1) {
        perror("poll");
        exit(1);
    }

    // Run through the existing connections looking for data to read
    for(int i = 0; i < fd_count; i++) {
```

Look at 'Slightly Advanced Techniques' in the Beej's guide

How can sockets be used to communicate between two processes on the same machine?