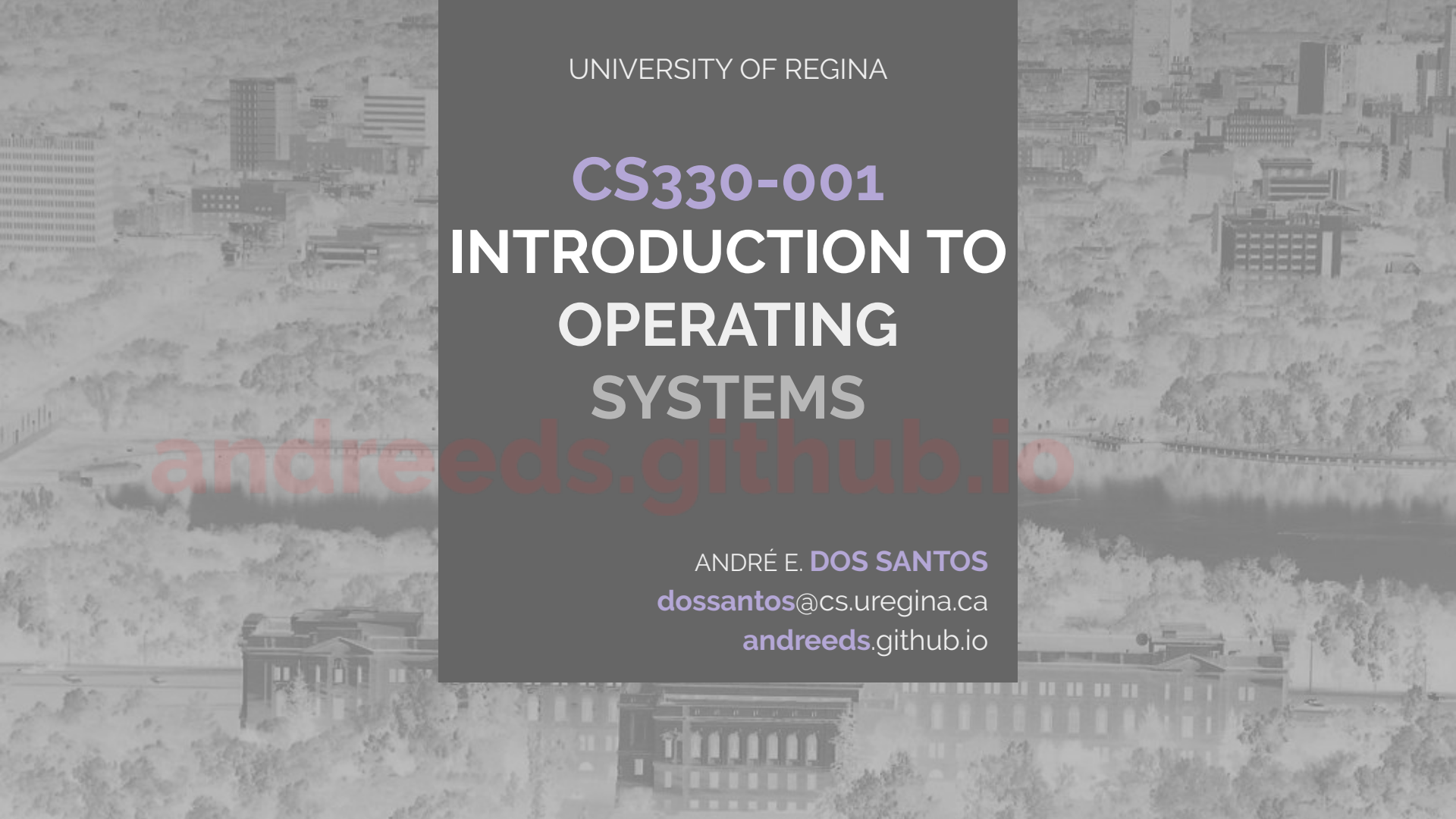


UNIVERSITY OF REGINA

CS330-001
INTRODUCTION TO OPERATING SYSTEMS

ANDRÉ E. **DOS SANTOS**
dossantos@cs.uregina.ca
andreeds.github.io

CS330-001
INTRODUCTION TO
OPERATING SYSTEMS

PROGRAMMING SOCKETS

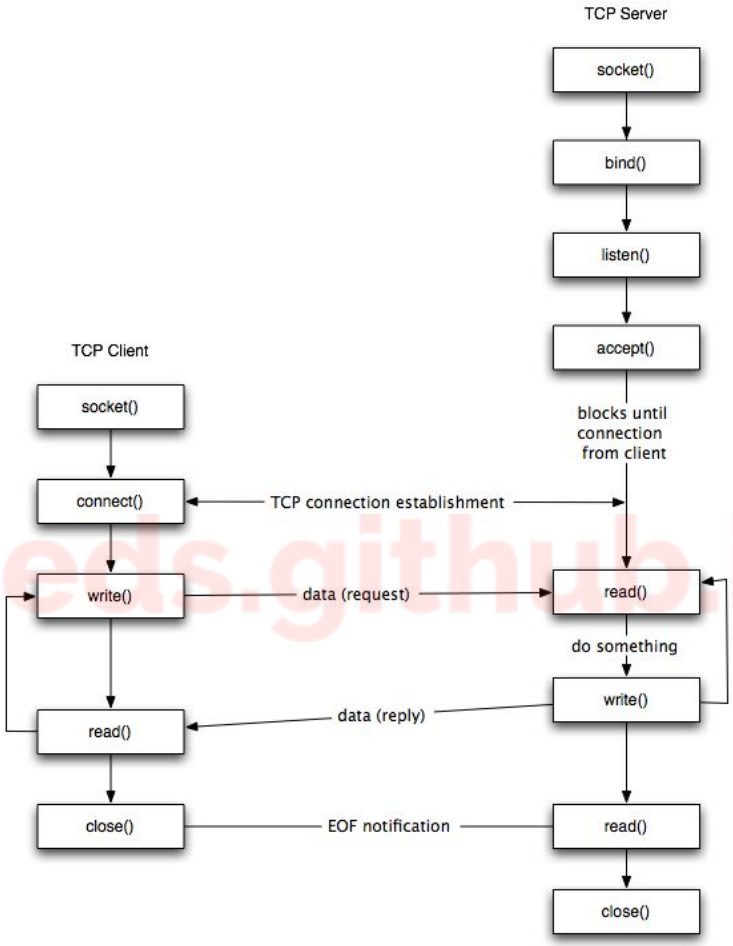
C

ANDRÉ E. **DOS SANTOS**
dossantos@cs.uregina.ca
andreeds.github.io

PROGRAMMING SOCKETS

- BSD sockets
  - a fundamental component in implementing applications using the message passing paradigm on UNIX systems
- BSD sockets (API) available to programmers on UNIX systems

- A socket is an endpoint used by a process for bidirectional communication with another socket/process, usually across a network
- Generally, a socket is described by an IP address and a port number



SOCKET ADDRESSES

- Many of the networking system calls require a pointer to a socket address structure as an argument

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

struct sockaddr
{
   u_short sa_family;     //address family
   char    sa_data [14];  //up to 14 bytes of protocol-specific address
};
```

SOCKET ADDRESSES

- For Internet protocol-specific addresses, an Internet address structure is required

```
#include <netinet/in.h>

struct in_addr
{
   u_long s_addr;      // 32 bit net id / host id
};

struct sockaddr_in
{
   short           sin_family;      // AF_INET
   u_short         sin_port;        // 16 bit port number
   struct in_addr  sin_addr;        // 32 bit net id / host id
   char            sin_zero [8];    //  unused
};
```


- The data type definitions (i.e., typedefs) for `u_short` and `u_long` are provided in `<sys/types.h>`

ELEMENTARY SOCKET SYSTEM CALLS

A **connection** describes the **association** (i.e., communication link) **between two processes**

- An **association** is a 5-tuple that completely specifies the two processes that make up a connection:

  `{protocol, local-address, local-process, foreign-address, foreign-process}`

  - The `local-address` and `foreign-address` specify the **network ID** and **host ID** of the **local** and **foreign hosts**, respectively
  - The `local-address` and `foreign-process` identify the specific port on each end of the connection
- Example

  `{TCP, 192.43.235.2, 1500, 192.43.235.6, 21}`
- A **half association** is either
  - `{protocol, local-address, local-process}` or
  - `{protocol, foreign-address, foreign-process}`,
    - each describing one end of a connection (i.e., a socket)

SOCKET SYSTEM CALLS

The `socket` system call is used to **obtain a socket descriptor** representing an incompletely specified communication endpoint

```
#include <sys/types.h>
#include <sys/socket.h>
int socket (int family, int type, int protocol);
```

- For our purposes, we set
  - `family` to `AF_INET`
    - i.e., address family / Internet protocols
  - `type` to `SOCK_STREAM`
    - specifies sequenced, reliable, two-way, connection-oriented byte streams and is typically implemented with TCP
  - `protocol` to `0`
    - this is the default since the type `SOCK_STREAM` has only one protocol available, namely, `TCP`
- If successful, `socket` returns a non-negative integer corresponding to a **socket descriptor**
- If unsuccessful, `socket` returns **`-1`** and sets `errno`
- **The socket system call fills in the `protocol` element of the association 5-tuple**

SOCKET SYSTEM CALLS

Example

- `socket` system call
  - both client and server

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

SOCKET SYSTEM CALLS

The `bind` system call is used by a **TCP server to associate the handle for a socket communication endpoint with a logical network connection**

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

- `sockfd` is the **socket descriptor** returned by a previous call to `socket`
- `myaddr` is a **pointer** to a variable of type `sockaddr` (structure)
- `addrlen` is the **number of bytes** in a `sockaddr` structure

- If **successful**, `bind` returns **0**
- If **unsuccessful**, `bind` returns **–1** and sets `errno`
- **The bind system call fills in the `local-address` and `local-process` elements of the association 5-tuple**

SOCKET SYSTEM CALLS

Example

- **bind** system call
  - **server only**

```
int socketFD;
struct sockaddr_in serverAddress;

bind (socketFD, (struct sockaddr_in *) &serverAddress, sizeof (serverAddress));
```

SOCKET SYSTEM CALLS

The `listen` system call is used by a **TCP server to indicate that it is ready to receive connection requests and to specify the number of waiting connection requests**

```
#include <sys/socket.h>
int listen (int sockfd, int backlog);
```

- `backlog` specifies how many connection requests can be queued by the system while it waits for the server to return from handling a previous request
  - i.e., the server needs time to fork a child process and return
- If successful, `listen` returns **0**
- If unsuccessful, `listen` returns **–1** and sets `errno`

- When a socket is created by the socket `system` call, it is assumed to be an **active socket**
  - server and client
- An **active socket** is expected to eventually issue a connect system call
  - i.e., to set up the bidirectional communication with another socket

- The `listen` system call converts an unconnected socket into a **passive socket**
  - i.e., it should accept incoming connection requests

SOCKET SYSTEM CALLS

Example

- `listen` system call
  - **server only**

```
int socketFD;
listen (socketFD, 1);
```

SOCKET SYSTEM CALLS

The `accept` system call is used by a **TCP server to complete the logical connection between the server and the client**

```
#include <sys/types.h>
#include <sys/socket.h>
int accept (int sockfd, struct sockaddr *client, int *addrlen);
```

- The `client` and `addrlen` parameters are used to return the address of the connected process
- The `accept` system call is used by a TCP server to **return the next completed connection from the head of the connection queue established by the listen system call**
  - Automatically creates a new socket descriptor
  - After a connection request is received and accepted, a server typically forks, with the child process servicing the connection on the new socket descriptor and the parent process waiting for another connection request
  - If there are no connection requests pending, `accept` blocks
- If successful, `accept` returns a **new socket descriptor**
- If unsuccessful, `accept` returns **–1** and sets `errno`
- The `accept` system call can fill in the `foreign-address` and `foreign-process` elements of the association 5-tuple in the child process

SOCKET SYSTEM CALLS

Example

- `accept` system call
  - **server only**

```
int socketFD;
int newSocketFD;
struct sockaddr_in clientAddress;
socklen_t addressLength;

newSocketFD = accept (socketFD, (struct sockaddr_in *) &clientAddress,
&addressLength);
```

SOCKET SYSTEM CALLS

The `connect` system call is used by a TCP client to establish a link to the known port of the server

```
#include <sys/types.h>
#include <sys/socket.h>
int connect (int sockfd, struct sockaddr *servaddr, int addrlen);
```

- `sockfd` is the socket descriptor returned by a previous call to socket
- `servaddr` is a pointer to a variable of type `sockaddr` (structure)
- `addrlen` is the number of bytes in the `sockaddr` (structure)

- The client does not have to bind to a local address before a call to `connect`
- The `connect` system call fills in the `local-address`, `local-process`, `foreign-address`, `foreign-process` elements of the association 5-tuple

- If successful, `connect` returns **0**
- If unsuccessful, `connect` returns **–1** and sets `errno`

SOCKET SYSTEM CALLS

Example

- `connect` system call
  - **client only**

```
int socketFD;
struct sockaddr_in serverAddress;

connect (socketFD, (struct sockaddr *) &serverAddr, sizeof (serverAddr));
```

OTHER USEFUL FUNCTIONS

Byte Manipulation

- There are multibyte fields in the various socket address structures that need to be manipulated
- These multibyte fields may not necessarily be standard `C`-strings (i.e., null terminated)

```
#include <string.h>
```

```
memcpy (char *dest, char *src, int bytes);
```

- `memcpy` moves the specified number of bytes from the source to the destination

```
memset (char *dest, int ch int bytes);
```

- `memset` writes the character ch (as an **`unsigned char`**) into the specified number of bytes in the destination

```
int memcmp (char *ptr1, char *ptr2, int bytes);
```

- `memcmp` compares two arbitrary bytes strings, returning **0** if the strings are equal, a value greater than **0** if `*ptr1` is greater than `*ptr2`, and a value less than **0** if `*pt` is less than `*ptr2`

OTHER USEFUL FUNCTIONS

Byte Ordering

- There may be differences in byte order between different computer architectures and different network protocols
- The following four functions can be used to handle the conversions required.


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

u_long htonl (u_long hostlong);      // host-to-network long
u_short htons (u_short hostshort);   // host-to-network short
u_long ntohl (u_long netlong);       // network-to-host long
u_short ntohs (u_short netshort);    // network-to-host short
```

OTHER USEFUL FUNCTIONS

Address Conversion

- An Internet address is usually written in dotted-decimal format (e.g., `192.43.235.1`)
- The following two functions are used to convert between dotted-decimal format and that contained in an in_addr structure.


```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
unsigned long inet_addr (char *ptr);
```

- `inet_addr` converts a character string in dotted-decimal format to a 32 bit Internet address

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

- `inet_ntoa` converts a 32 bit Internet address to dotted-decimal format

OTHER USEFUL FUNCTIONS

- We usually refer to computers by their human-readable names
- However, computers refer to computers by their numerical addresses
- Consequently, library routines are provided to convert between human-readable names and numeric addresses

OTHER USEFUL FUNCTIONS

```
#include <unistd.h>
int gethostname (char *hostname, int len);
```

- The returned parameter, `gethostname`, is null-terminated
- If successful, gethostname returns **0**
- If unsuccessful, gethostname returns **–1**

OTHER USEFUL FUNCTIONS

```
#include <netdb.h>
struct hostent *gethostbyname (char *hostname);
```

- The `gethostbyname` function returns a pointer to a hostent structure.

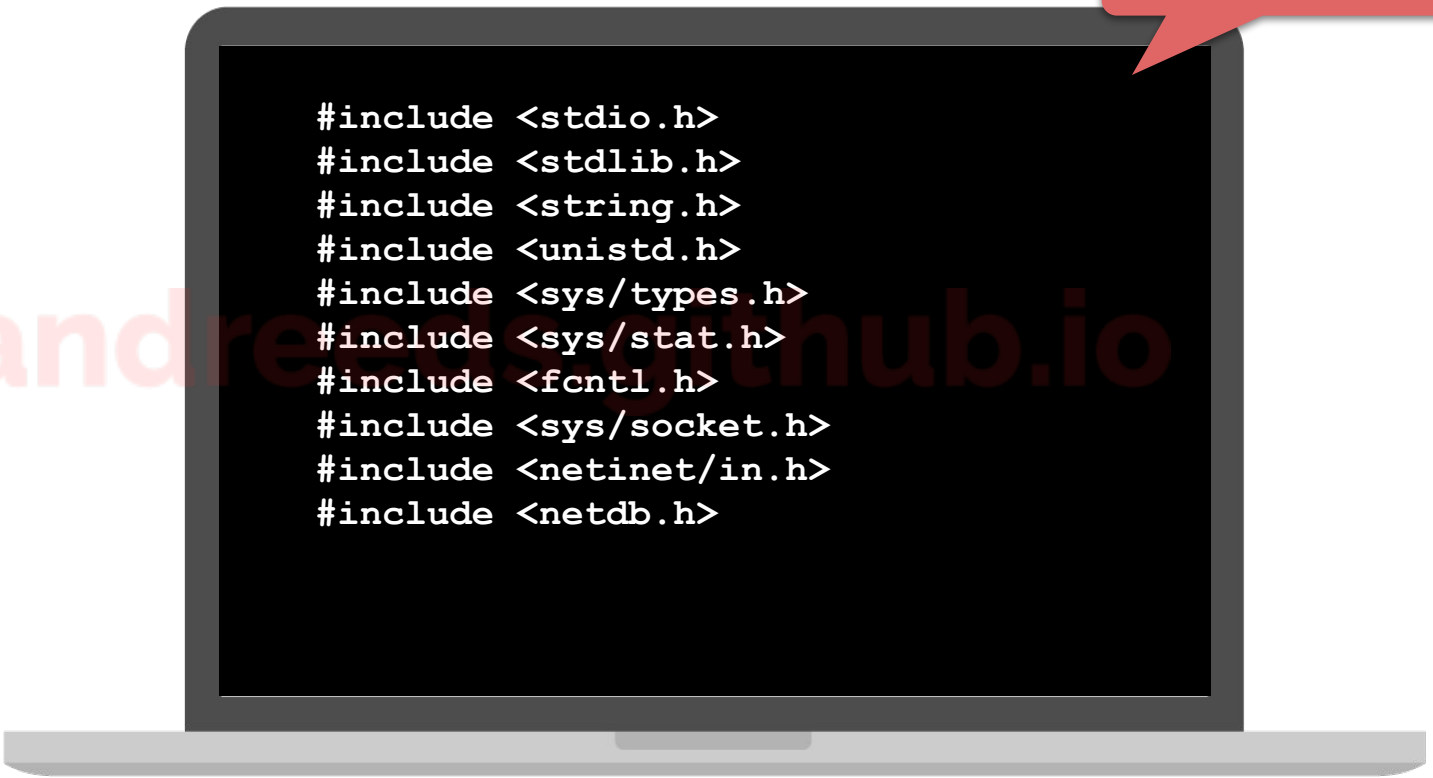
```
struct hostent
{
    char *h_name;           // official name of host
    char **h_aliases;
    int  h_addrtype;        // host address type (always contains AF_INET)
    int  h_length;          // length of address (always contains 4)
    char **h_addr_list;     // null terminated list of addresses from name server
};

#define h_addr h_addr_list [0]
```


- The `h_addrtype` field always contains `AF_INET`
- The `h_length` field always contains 4 (i.e., the length of an Internet address in bytes)
- The `h_addr_list` field is actually an array of pointers to structures of type `in_addr`

PIPE SYSTEM CALLS

1ServerExampleReadFile.c • 1ClientExampleReadFile.c

URCourses



PIPE SYSTEM CALLS

2socket[Client/Server/Functions/Include]Hilderman.[c/h]

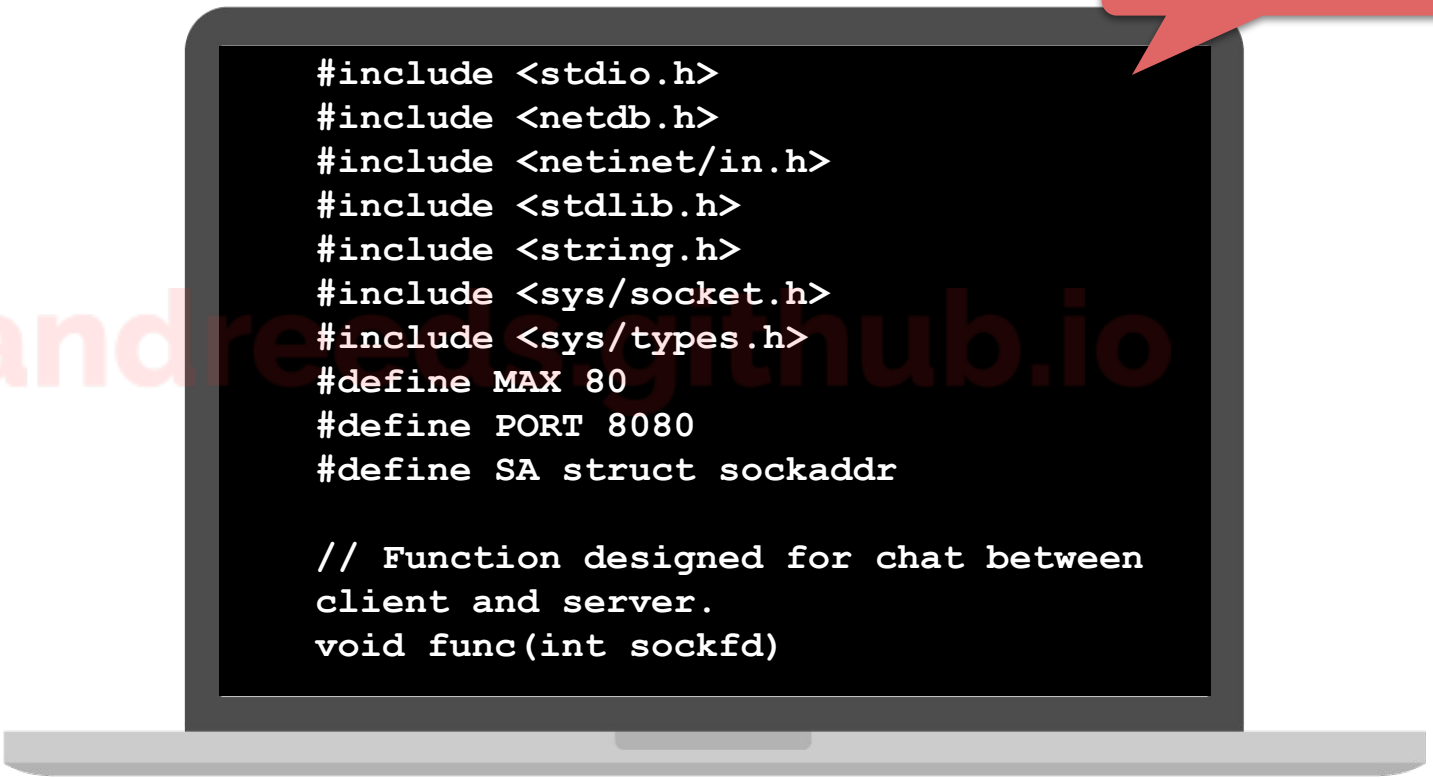
URCourses

```
#include "socketInclude.h"
#include "socketFunctions.h"

int main (int argc, char **argv)
{
    int i;
    int socket_fd;
    struct sockaddr_in server_addr;
    struct hostent *hp;

    if (argc < 2)
    {
```

PIPE SYSTEM CALLS

3TCP[Client/Server].c

URCourses



```
#include <stdio.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/types.h>
#define MAX 80
#define PORT 8080
#define SA struct sockaddr

// Function designed for chat between
client and server.
void func(int sockfd)
```