# COSC 3360 SECOND ASSIGNMENT

jfparis@uh.edu
Spring 2020

## The weather report

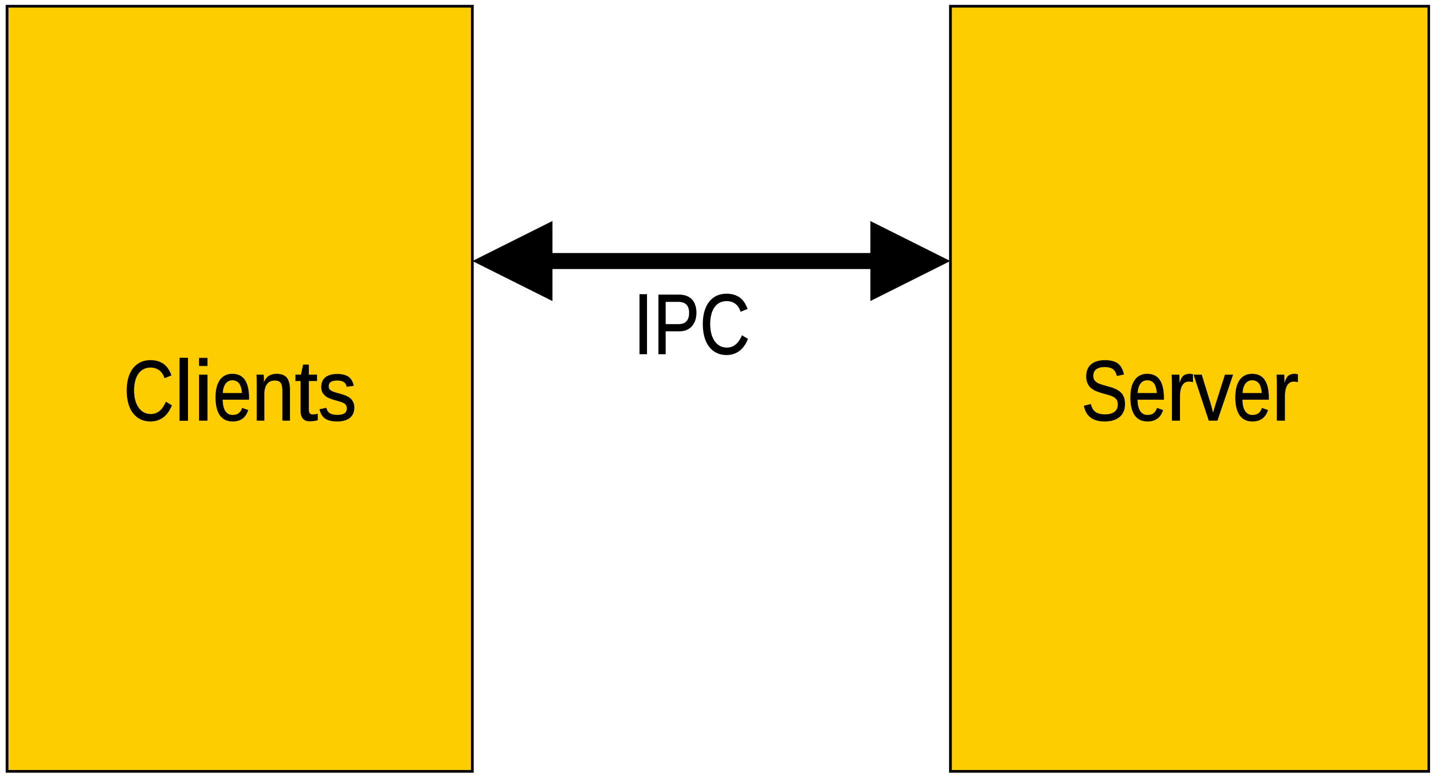
## The big idea

- Will build a simple client/server pair
- Server will maintain a table listing next day weather predictions for cities

  -
    ```
    El Paso,67,Sunny
    Houston,77,AM Thunderstorms
    …
    ```
  - Three fields separated by commas
- Clients will query the table

## YOUR PROGRAM

- Two parts



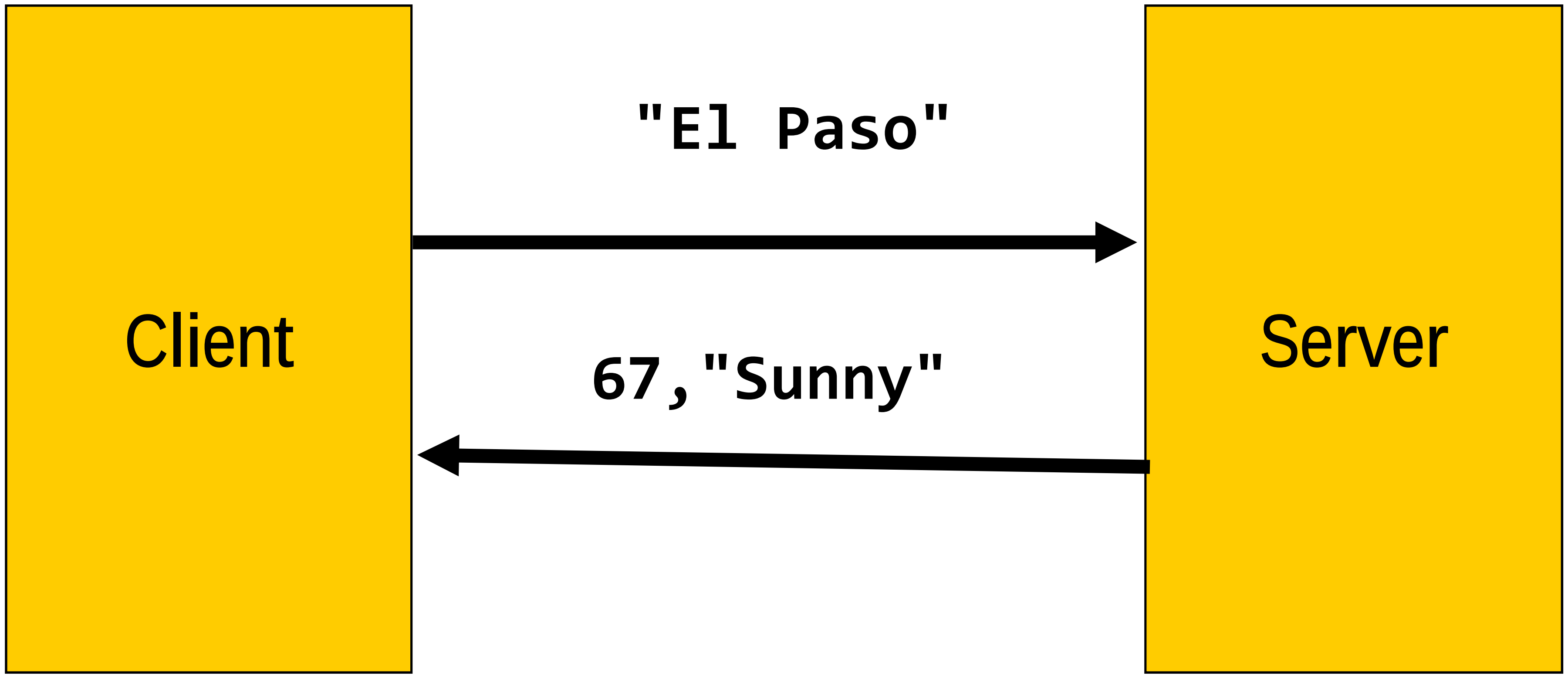
## In more detail: The client

- Prompts the user for the server's host name and port number
- Prompts the user for a city name
- Sends that name to the server
- Waits for reply containing the corresponding daily maximum temperature and sky condition
- Displays them to the user

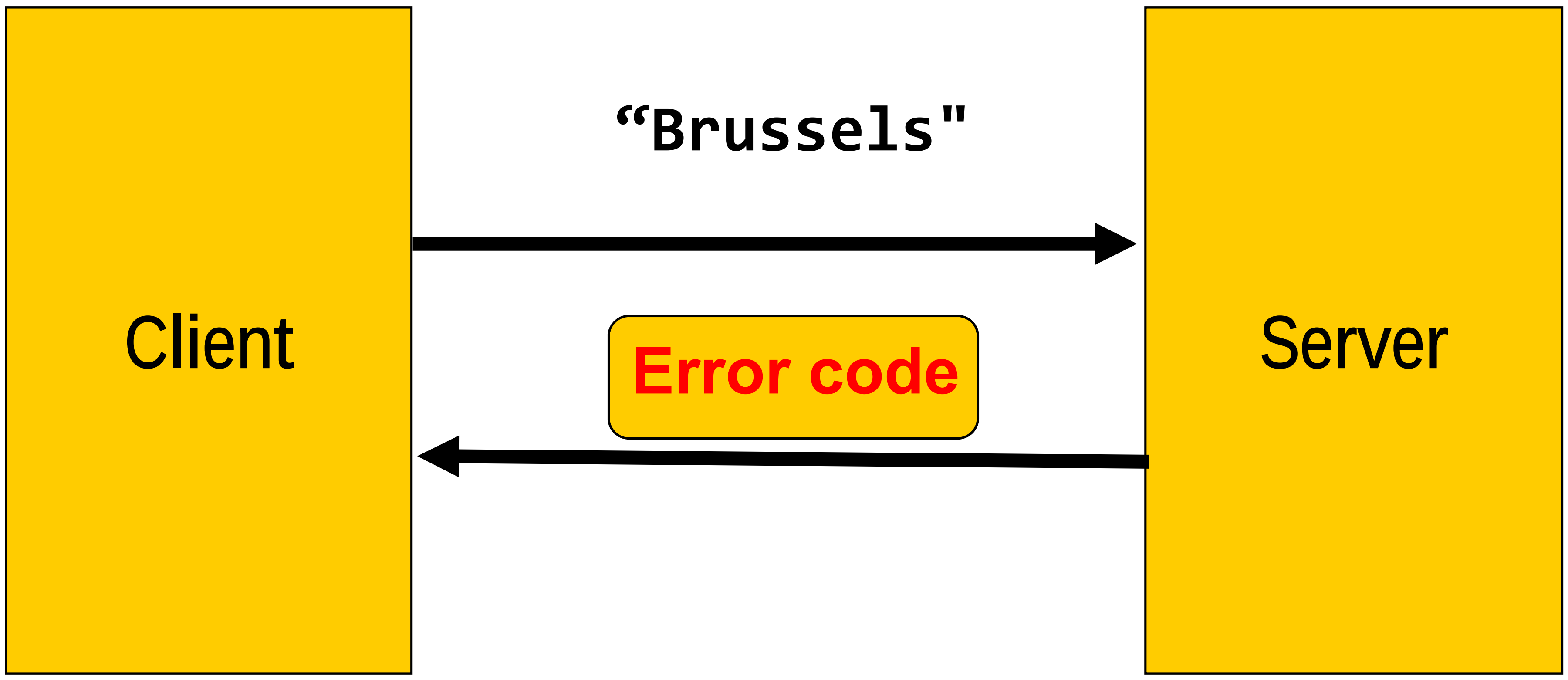
## In more detail: The server

- Single-threaded server
- Stores city names, corresponding daily maximum temperature and sky conditions in an ***in-memory*** table
- Prompts the user for a port number
- Repeatedly
  - Waits for a request
  - Answers it by sending the requested data

## The messages being exchanged



## The messages being exchanged



## The client

- Client will :
  1. Prompt user for server's host name and port number
  2. Prompt user for a city name
  3. Create a socket
  4. Connect it to the server
  5. Send the city name to the server
  6. Wait for the corresponding weather data
  7. Close the socket
  8. Print out what it got from the server

## Phone analogy

- Client will:
  1. Prompt user for an area code and phone number
  2. Prompt the user for a city name
  3. Get a phone
  4. Call the server
  5. Tell the city name to the server
  6. Wait for a reply
  7. Hang up
  8. Print out the answer of the server

## Server side

- Server will:
  1. Create a socket
  2. Bind an address to that socket
  3. Set up a buffer size for that socket
  4. Wait for incoming calls
  5. Accept incoming calls
     (and get a new socket)
  6. Reply with the requested weather data
  7. Hang up
  8. Return to step 4

## Phone analogy

- Server will

1. Get a phone
2. Get a phone number
3. Wait for incoming calls
4. Accept incoming calls  
   (and transfer them to a new line)
5. Listen to what the client says
6. Reply with the requested weather data
7. Hang up
8. Wait for new incoming calls

## Communicating through sockets

## TCP socket calls (I)

- **<u>socket(…)</u>**
  creates a new socket of a given socket type (*both client and server sides*)
- **<u>bind(…)</u>**
  binds a socket to a socket address structure ***(server side***)
- **<u>listen(…)</u>**
  puts a bound TCP socket into listening state (***server side***)

## TCP socket calls (II)

- **connect(…)**
  requests a new TCP connection from the server (***client side***)
- **accept(…)**
  accepts an incoming connect request and creates a new socket associated with the socket address pair of this connection
  (***server side***)

## Accept "magic" (I)

- **accept ()** was designed to implement multithreaded servers
  - Each time it accepts a connect request it creates a ***new socket*** to be used for the duration of that connection
  - Can, if we want, fork a child to handle that connection
    - ***Would not be necessary this time***

## Accept "magic" (II)

**Lets a child process do the work**

## TCP socket calls (III)

- **write()**
  sends  data to a remote socket
  (*both client and server sides*)
- **read()**
  receives data from a remote socket
  (*both client and server sides*)
- **close()**
  terminates a TCP connection
  (*both client and server sides*)

Apply to **sockets** as they do to **file descriptors**

## TCP socket calls (IV)

- **gethostbyname()**
  returns host address structure associated with a given host name

If your client and your server are on "well connected" computers, they will do:

```
gethostname(myname, MAXLEN);
hp = gethostbyname(myname);
```

## For personal computers

- Typically lack an internet host name
  - `jfparis@Odeon:~$ hostname`
    `Odeon`
- `Contrast with`
  - `-bash-4.2$ hostname`
    `program.cs.uh.edu`
- Easiest solution is to use **`localhost`** as client and server host names

| **Client side:** | | **Server side:** |
|---|---|---|
| `csd = socket(…)` | | `ssd = socket(…)` |
| | | `bind(…)` |
| | | `listen(…)` |
| `connect(csd, …)` | | `newsd = accept(…)` |
| `write(csd, …)` | → | `read(newsd, …)` |
| `read(csd, …)` | ← | `write(newsd, …)` |
| `close(csd)` | ↔ | `close(newsd)` |

## The connect/accept handshake

- For the connect/accept handshake to work, the user must specify the
  - host address  (**sa.sin_family**)
  - port number (**sa.sin_port**)

  of the server in its **connect( )** call

## Bad news and good news

- The bad news is that socket calls are somewhat esoteric
  - Might feel you are not fully understanding what you are writing

- The good news is most of these mysterious options are fairly standard

## Some examples (I)

-
  ```
  // create socket
  if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
      return(-1);
  ```

- With datagram sockets (SOCK_DGRAM), everything would be different
  - No **`listen()`**, **`accept()`**, **`connect()`**
  - Only **`sendto()`** and **`recvfrom()`**
  - Message boundaries would be preserved

## Some examples (II)

```
// SERVER ONLY
// declare sockaddr_in structure
struct sockaddr_in saddress
// get the name of your host
gethostname(myname, MAXHOSTNAME);
// get host address structure
hp= gethostbyname(myname);
//  set host address type
saddress.sin_family= hp->h_addrtype;
// set port number
saddress.sin_port= htons(portnum);
```

## Some examples (III)

```
// (Continuation of previous slide)
//bind address saddress to socket s
if (bind(s, &saddress,
    sizeof(struct sockaddr_in)) < 0) {
    // failure
    close(s);   return(-1);
} // if
```

## Picking a port number

- Your port number should be
  - ***Unique***
    - *Should not interfere with other students' programs*
  - ***Greater than or equal to* 1024**
    - *Lower numbers are reserved for privileged applications*

## Some examples (IV)

-
  ```
  // SERVER ONLY
  // set buffer size for a bound socket
  listen(s, 3);
  ```

-
  ```
  // SERVER ONLY
  // accept a connection
  int new_s;
  if ((new_s = accept(s, NULL, NULL)) < 0)
       return(-1) // cannot create new_s
  }
  ```

***The new socket***

## Some examples (V)

```
// CLIENT ONLY
// request a connection
// sa must contain address of server
// same code as before bind in server
if (connect(s, &sa, sizeof sa) < 0) {
    close(s);
    return(-1);
}
```

## Some examples (VI)

***A <u>fixed</u> number of bytes***

-
  ```
  // send a message
  write(s, buffer, nbytes);
  ```

-
  ```
  // read a message
  read(s, buffer, nbytes)
  ```

***The number of bytes read by the receiver must be equal to the number of bytes sent by the server***

## Doing networking assignments on your PC

- The host name of a Windows PC does not include its domain

  -
    ```
    jfparis@Odeon:~$ hostname
    Odeon
    ```
- `hp = gethostbyname("Odeon");` does ***not*** work
- Use instead **`localhost`**

  ```
  hp = gethostbyname(localhost);
  ```

## Implementation details

## The data table

- Read in from **weather20.txt** by the server
- Will contain city names, daily maximum temperatures, and sky conditions

  -
    ```
    Galveston,69,Thunderstorms
    Houston,77,AM Thunderstorms
    San Antonio,79,Mostly Cloudy
    …
    ```
  - Fields will be separated by commas
    - Strings can contain spaces

## The small details

- All your messages should ***either***
  - Have fixed sizes
  - Start by an integer occupying a ***fixed number*** of bytes and announcing the length of the remainder of the message

## Two good tutorials

- http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html
- http://www.cs.uh.edu/~paris/3360/Sockets.html
  - *Will also be on  the course Piazza page*