COSC 3360/6310 SECOND ASSIGNMENT

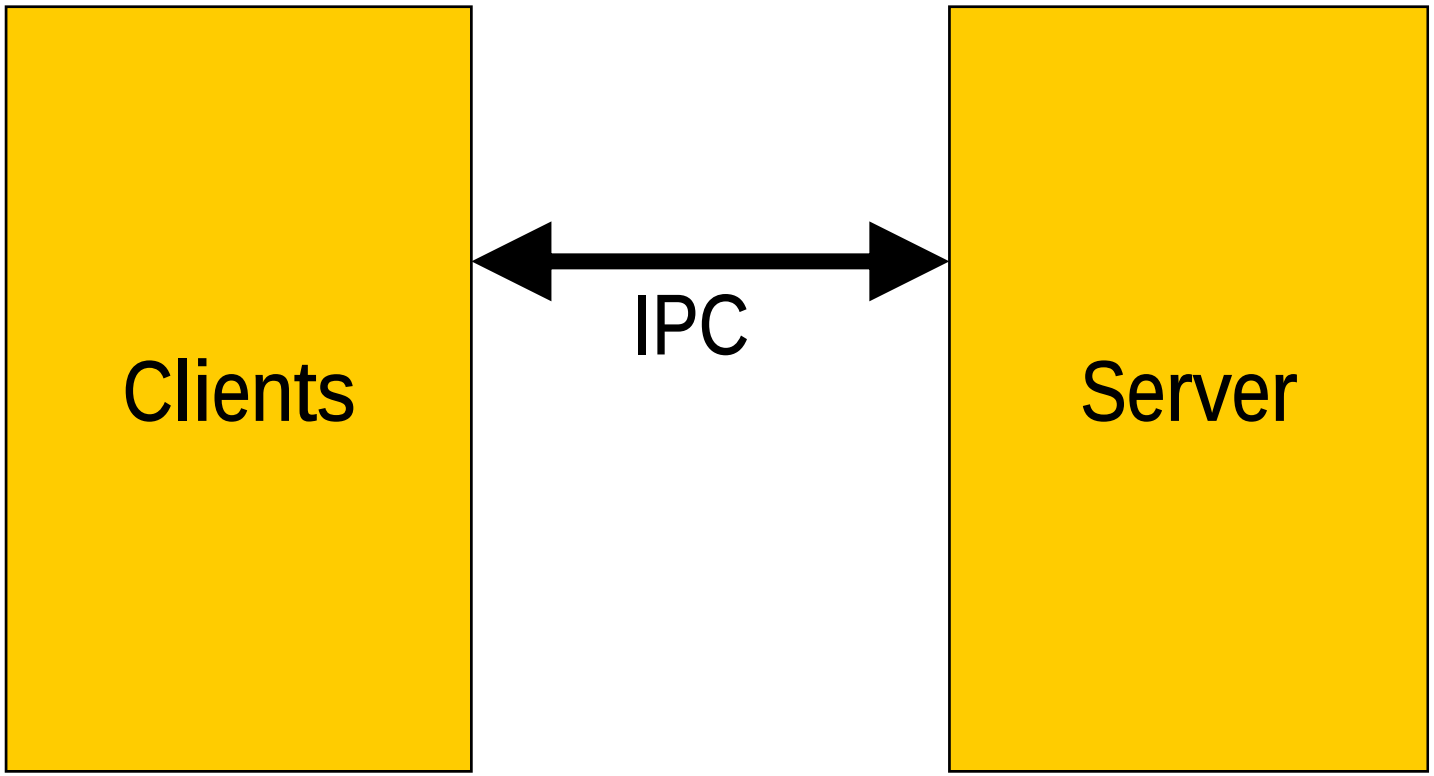
jfparis@uh.edu
Summer 2017

The big idea

- Will build a simple client/server pair
- Server will maintain a table listing the average sale price of a house for each of Houston neighborhoods:
  - **Acres Home, 123910**
    Addicks/ Park Ten, 210431
    Afton Oaks/ River Oaks, 1645821
    . . .
- Client will query the table

YOUR PROGRAM

- Two parts



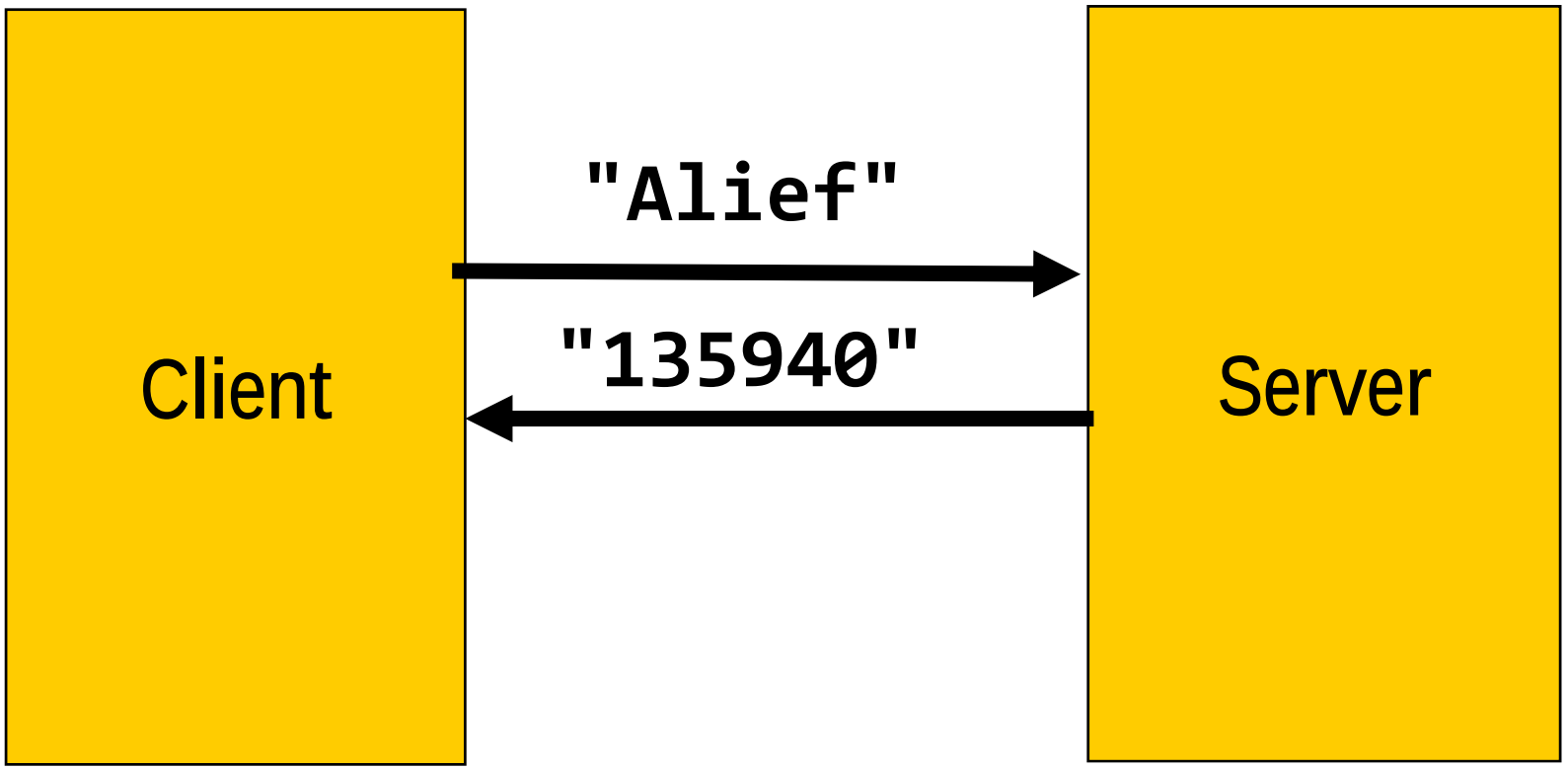
In more detail: The client

- Prompts the user for the server's host name and port number
- Repeatedly
  - Prompts the user for a neighborhood
  - Requests the average house price for that neighborhood from the server
  - Displays the result to the user
- Ends loop when the user enter an empty string

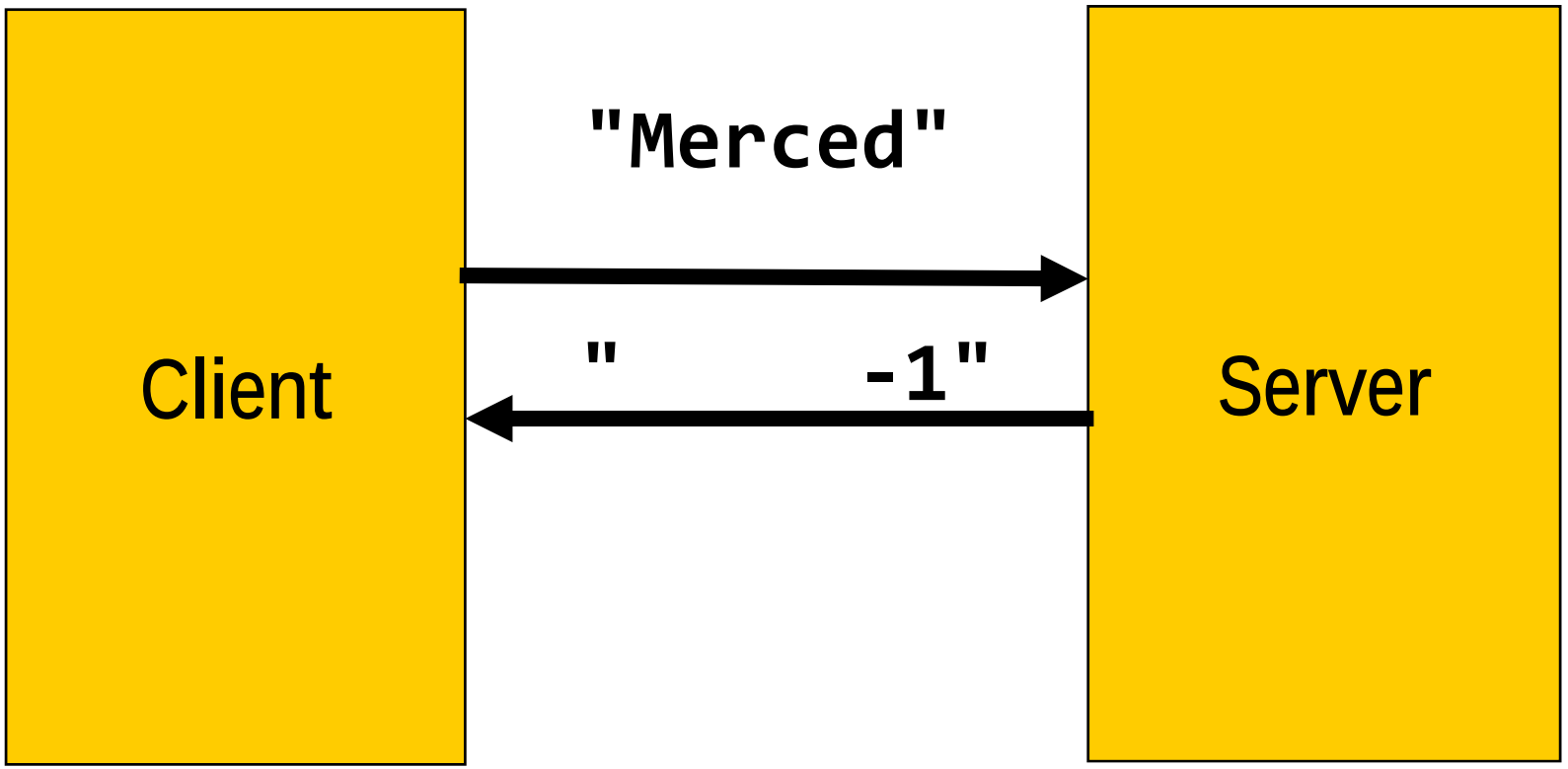
In more detail: The server

- Single-threaded server
- Stores user names and public keys in an in-memory table
- Prompts the user for a port number
- Repeatedly
  - Waits for a request
  - Answers each of them by sending the requested price

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 neighborhood name
  3. Create a socket
  4. Connect it to the server
  5. Send the user name to the server
  6. Wait for the house price
  7. Close the socket
  8. Print out the price it got from the server
  9. Return to step 2

Phone analogy

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

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 public key
  7. Hang up
  8. Return to step 2

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 client says
  6. Reply with the requested public key
  7. Hang up
  8. Wait for new incoming calls


Communicating
through sockets

TCP socket calls (I)

- **socket(…)**
  creates a new socket of a given socket type (*both client and server sid*es)
- **bind(…)**
  binds a socket to a socket address structure ***(server side***)
- **listen(…)**
  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)

Could let 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

Your client and your server will both be on the same host and you will do:

```
gethostname(myname, MAXLEN);
hp = get hostbyname(myname);
```

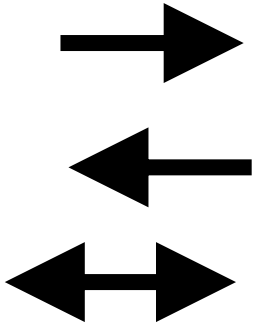
- **Client side:**

```
csd = socket(…)

connect(csd, …)
write(csd, …)
read(csd, …)
close(csd)
```

- **Server side:**

```
ssd = socket(…)
bind(…)
listen(…)
newsd =  accept(…)
read(newsd, …)
write(newsd, …)
close(newsd)
```



The connect/accept handshake

- For the connect/accept handshake to work, the user stub 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( ), no accept( ), no connect( )
  - Only sendto( ) and recvfrom( )
  - Message boundaries would be preserved

Some examples (II)

- // SERVER ONLY
  // get the name of your host
  gethostname(myname, MAXHOSTNAME);
  // get host address structure
  hp= **gethostbyname**(myname);
  sa.sin_family= hp->h_addrtype; // host address
  sa.sin_port= htons**(portnum**); // set port number
  //bind address sa to socket s
  if (bind(**s**, &**sa**, sizeof(struct sockaddr_in)) < 0) {
  close(s);
  return(-1);
  }

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 (III)

- // 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)/
  }

The reply socket

Some examples (IV)

```
// 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 (V)

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

A fixed number of bytes

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

Implementation details

The data table

- Read in from **`input2.txt`** by the server
- Will contain neighborhood names and average house prices.
- Up to 1,024 entries

Acres Home	**123910**
Addicks/ Park Ten	**210431**
. . .	. . .

The small details

- Neighborhood names will be short and but may contain spaces.
- A comma will separate neighborhood names from house prices.
- 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