PA1: CLIENT-SERVER INTERACTION

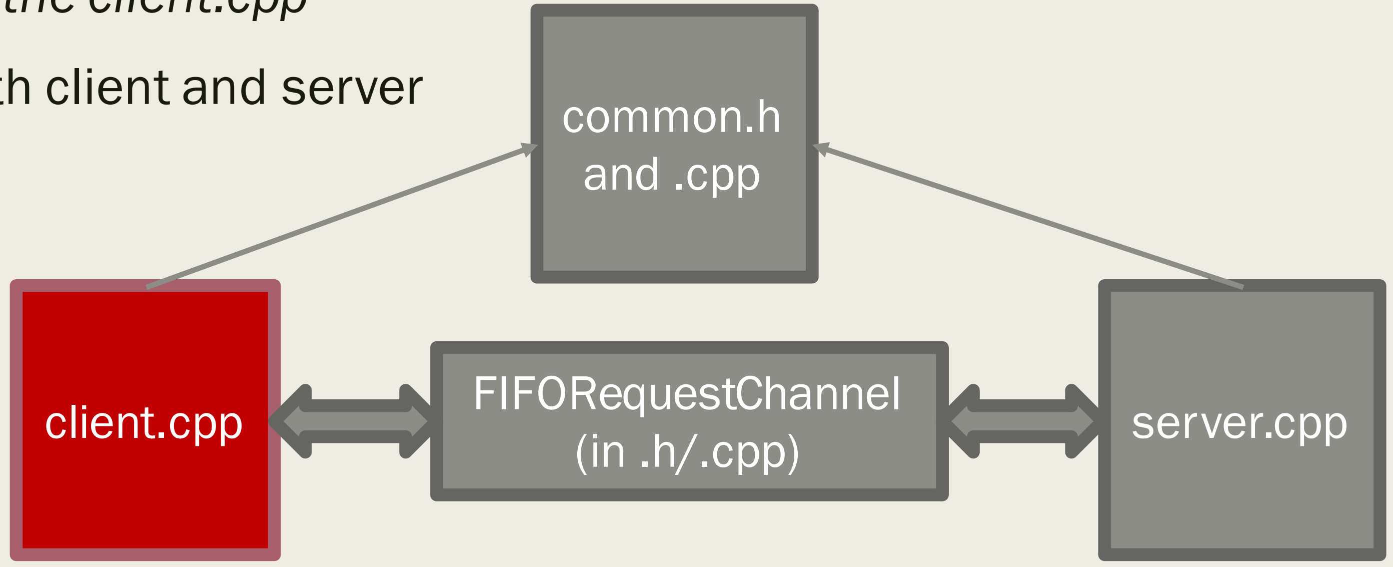
Fall 2020

Theme

- A server program hosts heart-rate data of some real patients
  - *It implements a communication protocol (i.e., request/response format) for different type of queries*
- Your task is to write a program that can collect these data by:
  - *Sending properly formatted message packets to the server*
  - *And then by receiving the response*

In a little more detail

- The server is like an Oracle – it has all the data and it knows how to answer when you “ask” it the right questions
  - *No need to change it or the underlying communication channel*
- The client does not know yet how to speak to this server
  - *You will instill this ability in the client.cpp*
- Common.h/.cpp are used by both client and server
  - *No change is needed there*



This is where all your code will go

Medium of Communication

- Client and server are 2 separate processes. To enable communication between the two, we need an Inter Process Communication (IPC) mechanism
- There are several IPC methods supported in linux
  - *FIFO pipes (aka, named pipes)*
  - *Message Queues*
  - *Shared Memory*
- We choose the first for its simplicity and speed, and implement that in FIFORequestChannel.h/.cpp
- To establish a connection between client and server, they have to agree on a name for the channel, which, let us assume is "tesla"
- Then, they have to write the following:

```
FIFORequestChannel chan ("tesla", Side);
```

  - *"Side" should be replaced by whatever side is making the connection*
  - *See client or server main() for example*

Talking Back-and-Forth

- Sending a message in either Client-Server or Server-Client direction is done by calling the FIFORequestChannel::cwrite () function
- Similarly, receiving in either direction is done by calling FIFORequestChannel::cread() function
- Cwrite() function takes 2 arguments:
  - *A char pointer to a message buffer (i.e., where you have stored the outgoing data)*
  - *An integer, indicating the number of bytes to send from the buffer*
- Cread() function also takes 2 arguments, the meaning are opposite:
  - *A char pointer to the message buffer where you expect to receive data*
  - *An integer indicating buffer capacity. This is a cautionary parameter to avoid any overflow from the other side*
- Look at either client.cpp or server.cpp for examples of usage

Server

- It contains heart-rate data for 15 patients – 1 csv file per patient under BIMDC/ directory
- Each file has ecg1 and ecg2 data over 60 seconds in 4ms increment totaling to 15K rows per file
- Each row has the following format:
  - *<timestamp as a double>, <ecg1 as a double>, <ecg2 as double>*
- The server can serve these data points one at a time in response to “datamsg” requests from the client
- In addition, the server supports 3 more types of requests. So, in total the server the following 4 types of messages:
  - *Data Message*
  - *File Message*
  - *New Channel Message*
  - *Quit Message*
- If you send some other message type, the serve simply sends back a error response whose type is: Unknown

Common Prefix

- You can find all message types in “common.h”
- Each message’s first field is an enum by the name MESSAGE_TYPE, also defined in common.h
  - *This common prefix is needed by the server to interpret the rest of the message*
- The other fields encode that type-specific data, as we will see in the following

Data Messages

- A data message is sent to the server to obtain a single data point
- Note that a data point is one ecg reading of a patient at a specific time stamp
- Therefore, a response to data message is always a double/floating point number
- From common.h, the definition of the corresponding class datamsg is as follows:
- For instance, if the client wants to get ecg2 data of patient 4 at time=10.004 sec, he just have to make a datamsg object with these values and send that through the channel
- In response, the server will send a single "double" value

```
class datamsg{
public:
    MESSAGE_TYPE mtype;
    int person;
    double seconds;
    int ecgno;
    datamsg (int _p, double _s, int _e){
        mtype=DATA_MSG, person=_p;
        seconds=_s, ecgno=_e;
    }
};
```

File Message

- The purpose is to ask for a raw file instead of any structured information
- First, to avoid server bog-down, you can only ask for a slice/window of the file
  - *The window size is limited by server's internal buffer size*
  - *Note that this buffer size comes as a runtime argument to the server with "-m"*
- Here is the class filemsg from common.h:
- The fields offset and length specify the window location and window size, respectively
- But there is no field for filename, which can be variable length
  - *You just have make a packet with filemsg + filename inside it, and then send it*
- Response: A char pointer to the buffer containing the specified window

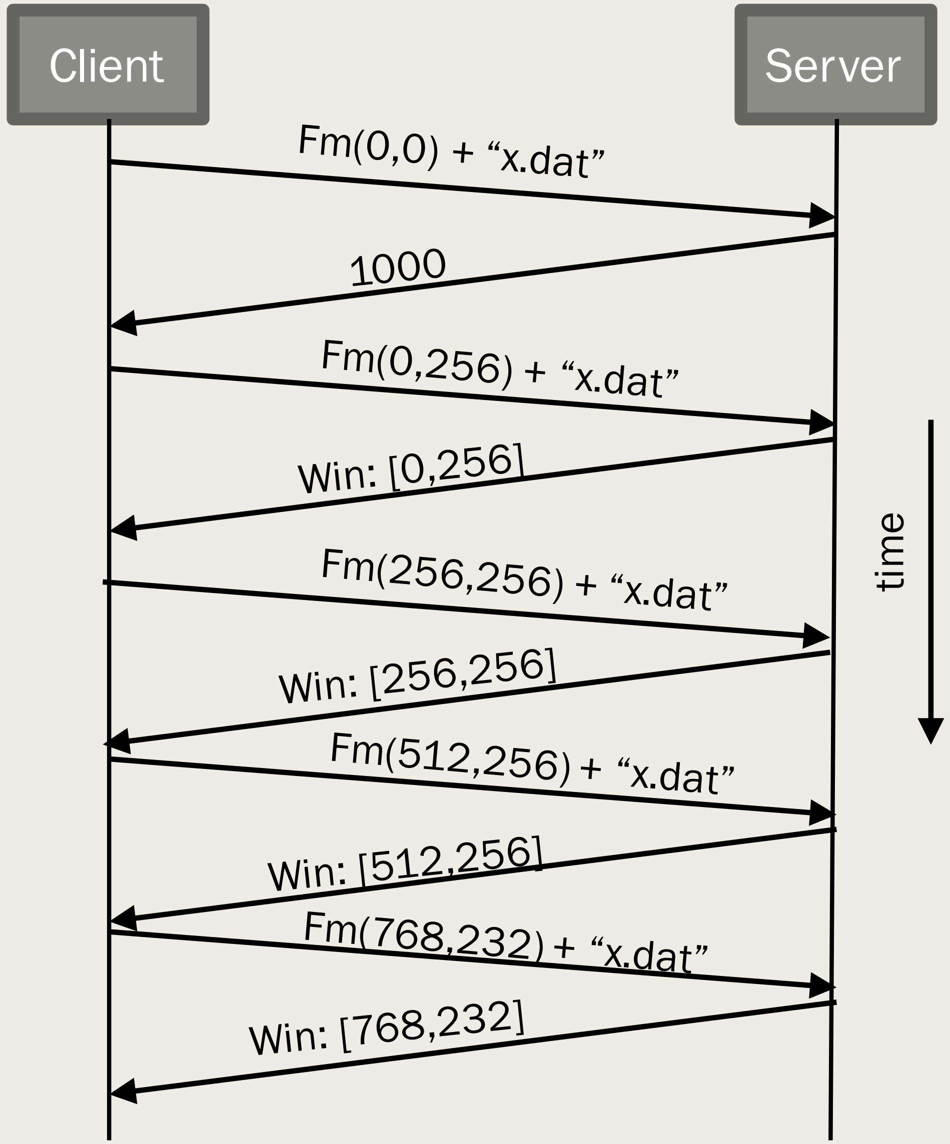
```
class filemsg{
public:
    MESSAGE_TYPE mtype;
    __int64_t offset;
    int length;

    filemsg (__int64_t _o, int _l){
        mtype = FILE_MSG, offset=_o, length=_l;
    }
};
```

Transferring a File

- If the client wants a full file, it has to do things:
  - *Know the length of the file*
  - *Based on the file length, come up with an array of requests to download the file in the form of windows as described in the previous*
- How to get the length of a file:
  - *Send a (filemsg + filename) message with filemsg(offset=0, length=0). This is a special request where the server responds with the file length as an __int64_t*
- The client then calculates the number of requests/responses it will need as: ceil (filelength/buffersize)
- Note that the client and the server need to use the same buffersize
  - *This can achieved by passing the same "-m" argument for both*
- Let us see an example

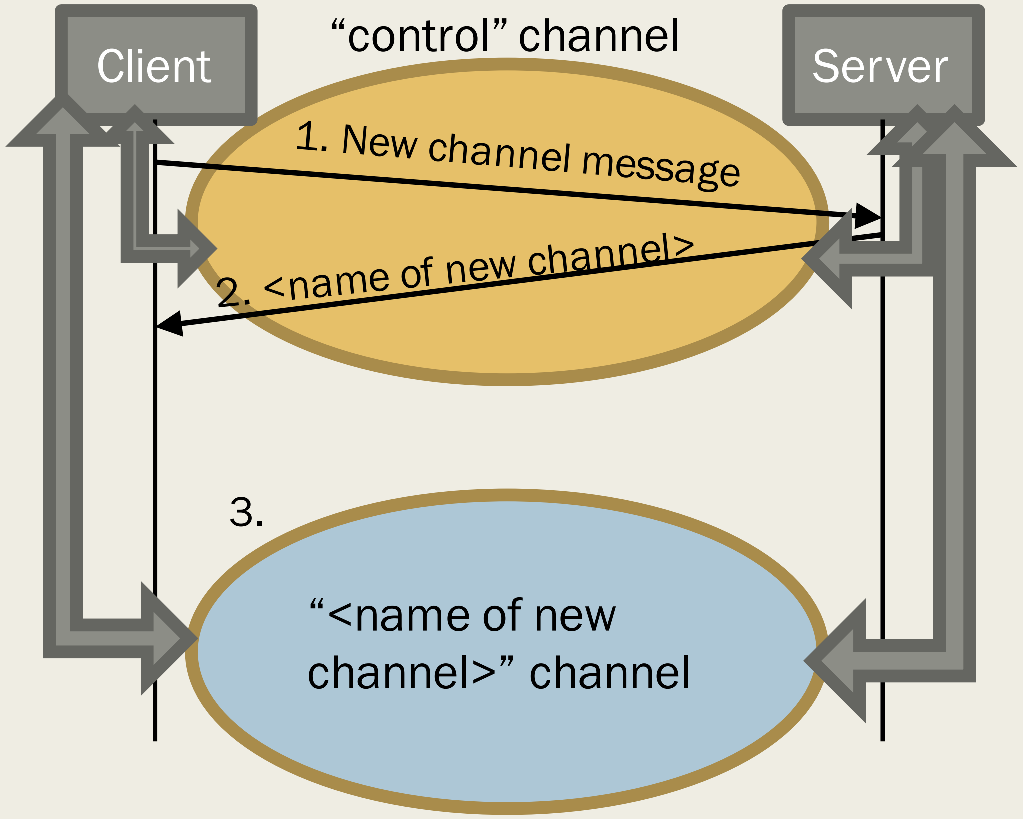
Example of File Transfer

- Assume the buffer size is 256 bytes for both client and server
- Say the file name is “x.dat” whose length is 1000 bytes
- To get the length, client sends a request (filemsg(0,0)+ “x.dat”)
- The server responds with 1000
- The client calculates that it needs ceil(1000/256) = 4 requests to get the file. They are:
  - *(filemsg(0,256)+ “x.dat”)*
  - *(filemsg(256,256)+ “x.dat”)*
  - *(filemsg(512,256)+ “x.dat”)*
  - *(filemsg(768,**232**)+ “x.dat”)*
- Note that the last request is smaller for 232 bytes
  - *You cannot specify 256 here because the server will catch it as an error*



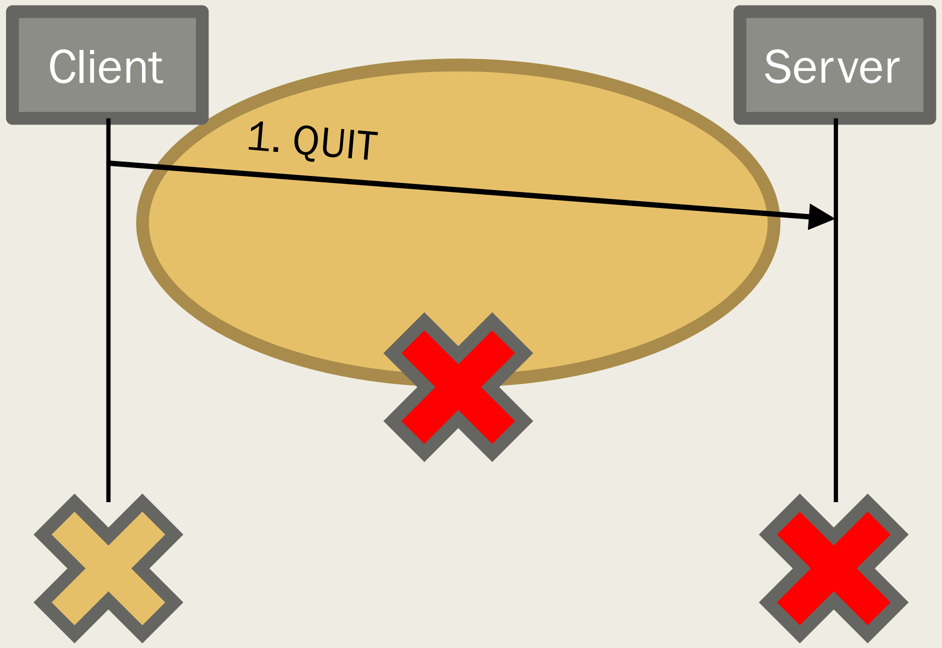
New Channel Message

- The purpose to create another simultaneous IPC channel between client and server
  - *This will be very useful in the later PAs when we want to speed up client-server communication by increasing bandwidth*
- This message is distinguished only by the message type prefix
  - *Unlike datamsg and filemsg, no need for a separate message type*
- However, there has to be a agreed-upon name for the new channel, that the server decides and sends back to the client
- Then, both the server and the client joins that new channel using the new name
- From that point on, all communication through the "control" channel and the new channel are independent and simultaneous



QUIT message

- Again, this message does not need anything other than the message type prefix which should be set to QUIT_MSG
- Note that this is the only message that the server does NOT reply to
- Both client and server terminates graciously after this message
  - *This message mean proper exit*
- A channel being closed without this message usually means an error
- The client needs to send QUIT for each channel
  - *Especially for channels created with "new channel message"*
  - *Otherwise the channel is not destroyed properly*



How to Run the given code

- Download the start_code in your env.
  - *Do not forget the BIMDC/ directory under the starter_code because that contains the data files (i.e., .csv files)*
- Open 2 terminals and navigate both to the downloaded directory
- Run “make” command from either terminal
- Then, run “./server” from one terminal and “./client” from the other
  - *You should not see much except both of them terminating*
- <u>Note for OSX terminal users:</u> Remove the “-lrt” flag from “makefile”
- <u>Note to Win+ VM users:</u> If you are using host-guest shared folders, you might see “cannot find file “fifo_control1” message. That means that Windows is not allowing you to create Linux-special fifo files in its own directories. The fix is not using a shared drive