Yu Yin Guan
yguan5@ucsc.edu
5/9/2014
CE 156
Section: Wednesday 4-6pm

Lab #3 Documentation

Introduction:

For this lab, I have created a reliable UDP client and server where the client is made concurrent with Pthreads while the server is made concurrent with the select function. The client and servers send messages back and forth to make sure of correct and clear transmission of files. In this document, I will discuss how I designed my program as well as some of the protocols and methods I used to make UDP reliable in detail.

Client:

For the client side of the program, I started out with the checking of the user input and declaration and initializations of variables that is needed for the program. One of these variables is the Pthread array which I contains the arguments needed for the pthreads to connect to the server. It contains stuff such as the filename of retrieval, pointers to write to, address of servers and so on. In the initialization, there is also this array called *connectFlag* which I used to indicate how many chunks I still have left that has yet to be obtained, it will be an key which allows the program to finish as my program will continue to loop until that array indicates that I have all the chunks of the file.

The diagram to my left is the flowchart at which is how my program is basically running. This diagram can be found under the Documentation folder. It describes my whole main loop and the rest minus the initialization part and some other details. In my main loop, I start off with a check of the amount of connects I have left to finish getting all the chunks of the files, since we need 1 connection per chunk (connects duplicates for servers if needed). If there are more chunks left than the amount of servers (counter), then set the number of connection for this loop (numCon) to be that amount. Otherwise, set it to the amount that is left. I then set the pointer to the server-info.text for a fresh start.

In the for loop, I get one line of server information, parse it to IP and port, create socket, set the server

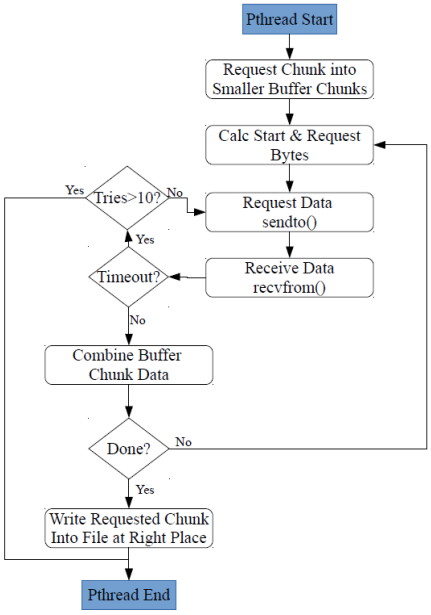
information for the socket and connect. In between I also set a timeout for the socket which is the function *setsockopt*. Since there is no function call connect, for UDP, I created a *sendto* and *recvfrom* in order to establish a connection. In this connect protocol, I send a packet containing the instruction "File Name : Name of the File", where the File Name part will indicates the server that this is a request for connection and it will look the argument, Name of the File, and look for this file by attempt to open it for reading. If the reading fails, I will expect the server to return an argument that says there is no such file. If everything goes well, The server will create a new socket with a new port, store my client's information and sends out a packet with "File Size : Size of File". Client will look at that as an Acknowledgement that the connection has been made and stores the files size for further processing.

In the event of not able to send to this server, the program will continue to the next server. For the case where the packet of connection request or the ACK is lost, I have the socket timeout to deal with that. So if any of those 2 packets is lost, recvfrom will not proceed, timeout will occur. When timeout occurs, I have a flag that indicates how many times in sequence this server has not responded to my packet, and if that flag is too high as in 10 consecutive lost, then I deem the server was unable to connect to. If the flag shows it has not tried for more than 10 times, it will send out a connect request again. So the whole process works as sending request of connection request and waiting for the ACK for, and resend request after timeout, up to 10 times before the server is seem to be unavailable for connection.

When the Client does establish the connection by receiving the ACK, the new address information which contains the new port given from the *recvfrom* function will be stored for Pthread uses. After all information is in the argument for Pthread, it will create a Pthread for this server and repeat this for all of the Servers., After that, it waits for them to join, update the success of retrieving the data in thread into the *connectFlag* array, updates the amount of chunks left still needing for transmission. The whole process The amount of Pthreads created will not exceed the amount of Servers available. So if a there are more chunks than servers, the process will repeats for the chunks left to transmit starting from the first server again and again until no chunks are left. If at anypoint the chunks are not transmitted correctly, pthread will indicate that in the *connectFlag* so the process will request for it again.

Pthreads:

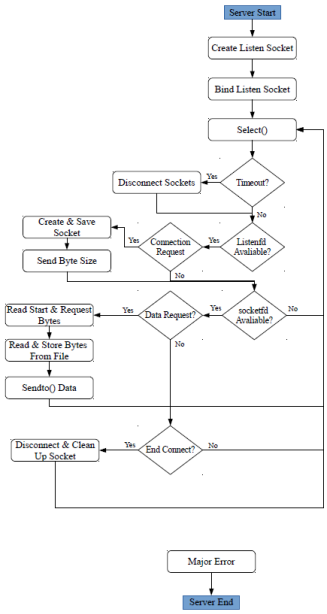
Pthreads are a part of the client side of code. The Pthreads handles the data transmission of chunks of files assigned to it. This is done so with a Start Byte and Request Bytes, which indicate which part of the file it should retrieve from server. If the Request Bytes is too big (1000Byte in my code) then split the request into multiple small chunks and request for those chunks in sequence. So there is a main thread loop where it first checks for the bytes left for retreival, parse the start and request bytes to get a small part of that Request Chunk of Bytes, then it sends request for that small part from server and waits for the replies. This is repeated until it has retrieved all data, if any part of this transfer has gone wrong as in it timeout too many times, or unable for retrieval, it will indicate on the *connectFlag* array that it has failed. It will exit and have the main loop to do a initial request for transfer again.



As shown on the picture on the left (from Documentation), the request for the small chunks has a timer to it. As in if the request or reply of the data has been lost, it will try again for 10 times and fail when the lost occurs for that small chunk happens consecutively for 10 times. Otherwise, it will combine all the small chunks it received and write it into the file after small chunks is retrieved. Each Pthread write to the file on the place it suppose to retrieve bytes for. To ensure the data of small parts are combined correctly, there is a window if it has retrived something that indicates a different small part start byte and request bytes then the one it just requested for, it will see it as failed and repeat request until it has retrieve that small part. So there is this sliding window for file transfer.

For a picture of how the whole client side of the code works in packet transfer which is ideal and have no packet lost, there is a picture in Documentation in the name of Packet_Transfer_Diagram that shows how I pictured it to be working in ideal situation with no lost.

Server:



Just to be clear, on the server side, there is no handling of reliability, all reliable control is handled by the client side. The server simply replies when it receive the correct request, and ignores ones that are not defined. The only control that might count as reliable control is the part where if no socket has said anything for a while, disconnect all sockets. I did that because if no request came in for a while in any socket, those sockets are probably dead or the server has died. Again, the picture to the left is a diagram of how the Server works, found in Documentation folder. The server is not suppose to end, but if something major went wrong like no socket can be created or bind anymore, or anything major in error, it will exit. Also, I left some debug mode where if enabled, it will delay the server to simulate a loss of packets for the client. There is description of the modes in the code.

The server starts off with declaration and initialization of the variables. It then create socket, set timeouts, address, bind listen socket, and put the listen socket in the fd_set for the select function. Then it goes to the main loop. The main loop starts with the reset of variables in the fd_set, buffers, and timeout times. It then goes to select and right under select is a handle on timeout where if it times out, it will disconnect all connections. I

currently have it at 10 seconds but it can be changed. After that it will go and check two major if statement that checks if the listen socket is available and another one for check all clients to see which one is available.

In the if statement of checking if listen socket is available, it first looks for the open slot in the client array, creates a socket, and initialize information . If the packet is a initialize packet for connection, it will look for the file request and bind socket. If file exist, it will send byte size and save this socket, otherwise, it will send no such file and skip saving this socket. After that, it will check if there are too many clients, update fd_sets, update amount of sockets, and skip the if check for client socket availability if all availability is handled.

In the Client socket check, it goes through all client sockets, checks to see if it is a end request. If it is an end request, close this socket, otherwise, do nothing. Then it checks if it is a request of data packet, if it is, it will parse the request to get the start byte and the amount to be read. It then goes to see if file is available for a double check, if it is not available then send no such file. Otherwise, open the file, set pointer to the start byte, read the amount requested, and send that data to the client then close the file.

Handling request is the only thing the server side does. All reliable checks and handles are done by the client, this way, server can handle clients faster because since we are not using threads for this, long codes in the handle might not be great. Also, handles on client side lets me gain more control.

server-info.text:

The server-info.text contains the IP and port of the list of servers you want to connect to for the file transfer. Each time there is a request of transfer, this file will be read multiple times until transfer is finished.

MakeFile:

The makefile compiles everything and outputs 2 files named Client and Server and some object files. The Client file is for the client side and the Server file is for the server side. Make clean will clean all the files created from the make command.

Result:

Resulting program is a UDP client/server where it will request for a connection, goes to the next server if that server is not available. After connection, pthreads will handle the transfer of data and flag the *connectionFlag* array if that chunk of the file is retrieved. The clients handles the timeout. When timeout occurs, it resents request and waits for response again, up to 10 times before it is deem disconnected. The pthread of client will make these chunks into smaller parts for UDP to transfer and assemble them into chunks of file. There is also timeout and a sliding window for these small parts to make sure all data is received, otherwise, flag *connectionFlag* to indicate it still need to be retrieve, in which case it might get it from another server. The client side will continuously to run until all chunks of the files are retrieved, and each chunk is written once they are properly received. This will result in a full

file once it is done with the connection. Also, after each Pthreads, it will close the connection made to the server. So in the whole main loop in client, it is always starting up new connections.