CPSC 3780 - Course Project Write-Up

Group Members: Everett Blakley, Mitchell Sulz-Martin, Dallan McCarthy

This document outlines the definition of the Protocol used to complete the Course Project for CPSC3780 Fall 2018.

Project Description
The goal of this project is to send simple ASCII text files across a network from a server to a number of clients. The project uses a custom protocol that defines how the client will request data and how the server will process the request and send data.

The Protocol works on a modified version of the Simplex Stop-and-Wait protocol discussed in lecture. The order of operations for this program is as follows:

1. Server sets up 2 sockets; one for data transmission, one for acknowledgement/negative acknowledgement.
2. The client creates client sockets on the same port numbers and with the server IP address as the host name
3. The client sends a “request” frame to the server
4. The server constructs the frames for the file requested and loads them into memory
5. The server puts the first frame into the socket, and waits for a response from the client
6. The client pull the frame out of the socket, performs a parity bit comparison, and sends an acknowledgement (`ACK`) or negative acknowledgement (`NAK`) frame back to the server accordingly
7. The server processes the response from the client, and either sends the next frame, or resends the previous frame
8. Repeat 5 to 7 until the entire file is transferred.
9. The client closes its socket and the program exits, while the server stays running

This project also runs under two scenarios: a simple one server - one client scenario, and a more complex one-server multiple-client scenario. In the latter, the server program will put each user into a new thread and process the threads simultaneously. If 6 clients request the server, the last client will have to wait until a thread is available. Once a thread opens up, it will be placed into that thread and process the request.

Project File Structure
The project files are organized into the following file structure:

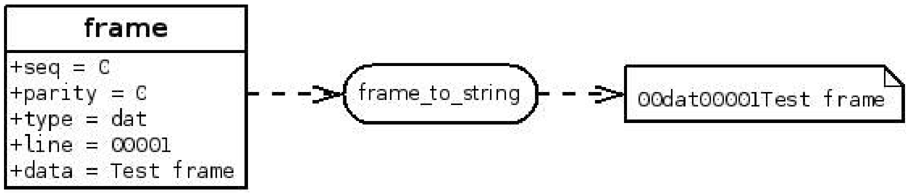
- root
    - src
        - ClientSocket.cpp
        - ServerSocket.cpp
        - Socket.cpp
    - include
        - ClientSocket.h
        - Protocol.h
        - ServerSocket.h
        - Socket.h
        - SocketException.h
    - S1
        - client
            - client.cpp
            - Makefile
        - server
            - files

        - Makefile
        - server.cpp
    - S2 (exact same files and structure as S1)
        - client
        - server

Protocol Description
The server will initialize two sockets, `serverAck` and `serverDat`, for transmitting `ACK/NAK` frames and `DAT/NAT` frames, respectively.
The protocol designed for this project uses a simple C++ struct, `frame`, to carry the data from the file. The frame struct has 5 data members:

Data member name	Type	Description
`seq`	`std::string`	Sequence number of the frame. Possible values: 0 or 1
`parity`	`std::string`	The even parity bit of the characters contained in `data`
`type`	`std::string`	The type of frame being transmitted. Possible frame types are: ● `ACK`: an acknowledgement that the all of the frame data is correct, and the server may send the next frame ● `NAK`: the frame was not correct, and the server must resend ● `DAT`: a data frame, containing data from the file requested ● `NAT`: not data, used to close the connection
`line`	`std::string`	The line number from the file that the `data` corresponds to. Up to six digits, i.e. less than 1,000,000
`data`	`std::string`	64 characters from the file

Since the sockets are designed to transmit string, the `frame_to_string` method will process the struct and convert it to a string for transmission. The following image shows the format of the strings that are sent via the sockets



With each frame being sent as a string, the program will reconstruct a frame struct using the `assemble_frame` method, which essentially does the opposite of the `frame_to_string` method.

Once the server is running both sockets, the client will make a request to the server for a file. This request is a `frame` placed in `socketAck` by the client. The only valuable information in this request frame is the `data` property, which will contain the name of the file the client is requesting.

With the request received, the server will then build all then call the `build_frames` method, which is designed to parse the data file and split it into 64 character portions, and build a frame for each portion of the file. Within the `build_frames` method, the server will call the `calcEvenParity` for the `data` attribute and then store that in the `parity` attribute for that `frame`. The sequence numbers are also calculated within `build_frames`, alternating between 0 and 1, beginning with 0. These frames are stored in a vector, and then the server will iterate through that vector to serve the data to the client. To test that the protocol correctly `NAK`s frames with incorrect parity, every 5th frame (or when `frameCtr % 5 == 0`) is given an incorrect parity bit. This is a way of simulating transmission errors, and ensures the protocol is properly equipped to deal with them.

Finally, now that the client has made its request, and the server has initialized the frames for the datafile corresponding to the request, the frame interchange can begin. The server will begin by placing the first frame in `serverDat`. The client will receive this frame, call `assemble_frame` to reconstruct a frame object from the string. There are two possible events:

1. The frame type is `DAT`, it contains data from the requested file. This corresponds to a separate sequence of possible events:
    a. If the sequence number does not match what the client is expecting, the client will send a `NAK`
    b. If the sequence number matches, but the parity is wrong, the client sends a `NAK`.
    c. If the sequence number and the parity are correct, the client will send and `ACK`
2. The frame type is `NAT`, meaning that the server has sent all of the data from the file. This will break the `while(true)` loop, and gracefully close the socket.

The client will continue to check the incoming `line` number from the frames, and once the `line` number changes, it will print all the data it has to the console. Additionally, the client will load the line into a text file. Essentially copying the file onto their disk.

On the server side, once the client places a frame in `serverAck`, the server will check if it is an `ACK`. If so, it sends the next frame. If it is a `NAK`, the server will recalculate the parity, and resend the frame. It will wait for the `ACK` and then move on. The server will continue to send frames until it reaches the end of the vector of frames.

Project Difficulties

Initially, we had difficulties understanding how sockets operate. Once we got that figured out, we played around to try to send data between two terminal windows. After successfully sending the “Hello world” message, we then tried framing a data file. This proved to be fairly simple, as we just had to chop the file into 64 character pieces.

After framing was complete, we then tried to build our first protocol, which would do all the sequence number and parity checks required. We got something working, and thought that we were in the clear. However, this was before we started purposely creating errors in our parity bit. Essentially, the protocol would work until it got an error and then it would catastrophically fail. Therefore, the protocol did not work. Many hours and a lot of head scratching later, we eventually got the protocol to properly identify when a frame should be `ACK`d and `NAK`d.

After the protocol was working properly, we had to implement the multithreading. This was initially difficult, but that’s to the gratuitous online community, we were able to find something that did what we needed it to do. Thankfully, since our protocol is fairly robust, there was nothing that needed to be changed from S1 to S2 on the protocol side. All that was changed was the server program to allow for multiple client requests at the same time.

All things considered, while this program was initially extremely frustrating, as we wrap up we are pretty proud with how much we were able to accomplish and what we learned along the way. That said, none of the group members have aspirations of network administration, though we now acknowledge how difficult the whole process actually is.