CPSC 3780
Project Report
Written By: Eric Den Haan and Adam Lefaivre

The first decision we had to make for the project was the programming language that we would use. We decided to use Python because of its excellent sockets interface and the ease with which rapid prototyping can be done. We used Python 2.7 in order to ensure that our project would run on the lab computers. We used the Python socket programming tutorial found on the IBM website (https://goo.gl/UIdzgx) to get started understanding the basics. This tutorial came with some good resources for the set-up of UDP client-server systems as well as the fundamental syntax for socket programming using the socket library in Python. With this info in mind we were able to set up the sockets for both the client and server programs, binding them to a port and address. To send and receive messages, we used the `sendto` and `recvfrom` function calls, in accordance with the fact that we needed to use UDP as the basis for our system. We chose to make it possible for the client program to choose which server it wanted to connect to, which meant we had to include the address and port information of each server in each client, but we could have easily just included one server's information in a client program for information hiding purposes.

From there, we wrote the basic skeleton for the client program first. We opted to use a system where the user would first be asked to enter a 10-digit ID number, and would then be presented with three options: press 1 to send a message, pressing 2 to send a get request, and pressing 3 to quit the program. Depending on which option was chosen, a different string would be constructed by the client program. We chose a total message size of 256 bytes as that was the size used in the sample given in the project description. We opted for a maximum message size of 160 characters, in line with the standard size for a text message. We chose to separate each field of the message with "`/`" characters, using the python `split` function to break the message apart into a `list` on the receiver side. The `split` function is handy as the user can specify how many times a string should be split (in our case, it was 4). This allowed us to ensure that "`/`" characters would still be allowed in the message payload. The user is first prompted to enter the 10-digit ID of the destination client, then the message payload contents. Upon receipt of a "send" type message, the client program calls the `split` function so that it can access the message fields by index, and displays the source and payload fields to the user.

One problem we noted near the end of our project implementation was the need to send multiple get requests from the client in order to receive any forwarded messages. This is due to a delay that occurs when the messages must first be forwarded to the correct server before they can then be delivered to that server's client. As a solution, we get the client program to send 5 get request messages every time the user opts to receive messages. Only if all 5 come back with an empty payload will the client tell the user that there are no messages for it. We realize that this may not be an optimal solution but it does work correctly for our purposes.

We also needed to implement two extra message types besides send, get, and ack: first, we use a "handshake" message when the client first connects to a server to make the server add the client to its list of clients and servers. The second type of message we implemented was a "terminate" message, which tells the server to remove the client from its list of clients and servers. This "terminate" message is sent when the user presses option 3 to quit the client program. In this way, our list stays as up to date as possible, allowing a smooth flow of messages across the network.

One of the first challenges when developing the basic functionality of the server was answering the question of who would take care of sequence numbers. We first implemented sequence numbers in the client program but soon realized that this would cause problems when handling messages within the server. As a result we have the server controlling the sequence numbers. We go from 0-99 and then roll the sequence number over to 0 again. Each time a "send" type message is processed, the sequence number increments by one. Just like in the client program, we made heavy use of the `split` function for breaking incoming strings into a `list` which can be accessed by index. In addition, when handling a get request and sending messages to a client, we used the `join` function to turn the list back into a string separated by "`/`" characters. Being able to access message fields by index makes it easy to make decisions about what kind of message is being received and who it is from, etc. Each time the program loop runs, we listen for a message and use a series of if statements to determine how the incoming message will be processed.

The real challenge in implementing the server program came in sharing the client-server list and forwarding messages between servers. We chose to use pairs containing a client ID and server IP address for our list. To insert these pairs into our list, we wrote a function `addClient` which checks to see if a client is already in the list attached to a server and, if it is not, inserts it. This function is called when a "handshake" type message is received by the server, as well as when a "send" or a "get" type message comes in. This ensures that if there is some kind of transmission error with the "handshake" message, we will still add the client eventually. A function `deleteClient` removes a client/server pair from the list when a "terminate" type message is received by the server. To send the list, we construct a string using "`/`" characters to separate client and servers within each pair, and "`?`" characters to separate each pair. This way, we can still use the `split` function to create `list` tuples on the receiver side. Two functions, `sendList` and `parseList` handle the routing. The former constructs the string containing the client/server list and sends it to the i+1 and i-1 servers. We had to ensure to take care of all cases, including where the server's index in the list of servers is 1 and when it is 5 as well as when it is in between. When the server's index is 1, it only sends to i+1, and when it is 5 it only sends the list to i-1. This avoids accessing invalid list indices. The `parseList` function processes an incoming client/server list string by inserting client/server pairs if new ones are found. The question of when to send the list was also a challenge. First, we were sending the list every time a "get" or "send" message came in to a server, but later we opted to send the list only when a "handshake" message came in as well as when the `addClient` function is called. This ensures that we are sending the list only when we need to to avoid clogging up the network with client/server lists. The server knows when it is receiving a client/server list message by checking to see if the third character in the incoming string is an integer. This works because none of the other message types will ever have this property. The `parseList` function also contains a flag which makes sure that there will be no feedback loops when the list is being shared around the network.

The only function that this list-sharing algorithm could not handle was the cases where a server was losing clients and needed to tell other servers to drop those clients from their client/server lists as well. We used a function called `forwardTerminates` to flood the received terminate message to every other server on the network. In addition, to avoid a feedback loop, we set a flag in the payload field of the "terminate" message. Once a server received an original "terminate", it would set the flag to 1, which would tell the servers reading a forwarded "terminate" to not also forward the message.

The final functionality we needed to implement was message forwarding between servers. We wrote a function called `forwardMessages` to do this. If a server received a message destined for a client not in its client/server list, it would add this message to a list called `messages_to_forward`, which is distinct from the regular list of messages to deliver to a server's clients. At each iteration of the program loop, we send this list of messages to forward if it is not empty. The function itself iterates through this list, finds the server the client belongs to, and then forwards the message to the correct server. Once the message has been forwarded, we remove it from our list of messages to forward.