

Florida Institute of Technology

MILESTONE 3

Client supporting UDP and TCP

Klaus Cipi, Peter Banis

School of Computing, Computer Science

25 March 2018

Introduction

This project is a simplified GOSSIP P2P system. The iterative server implemented in C supports commands from both TCP and UDP. Commands that can be handled by the iterative server are:

GOSSIP:[sha256 base64]:[time]:[message]% - Upon receiving this command, the server checks if it already knew it and simply discards it if it was known, while printing "DISCARDED" on the standard error. Otherwise stores it, broadcasts it to all known peers, and prints it on the standard error.

PEER:[name]:[port]:[ip]% - Upon receiving this command, the server stores the name and address of this peer if it is not yet known, or to updates its address.

PEERS?\n - Upon receiving this command, the server send the client all the peers that it has stored until the time of the request.

The client, which is used to send commands to the server, is implemented in Java and support both TCP and UDP.

Technologies Used

A list of all the technologies that were used to implement the iterative server:

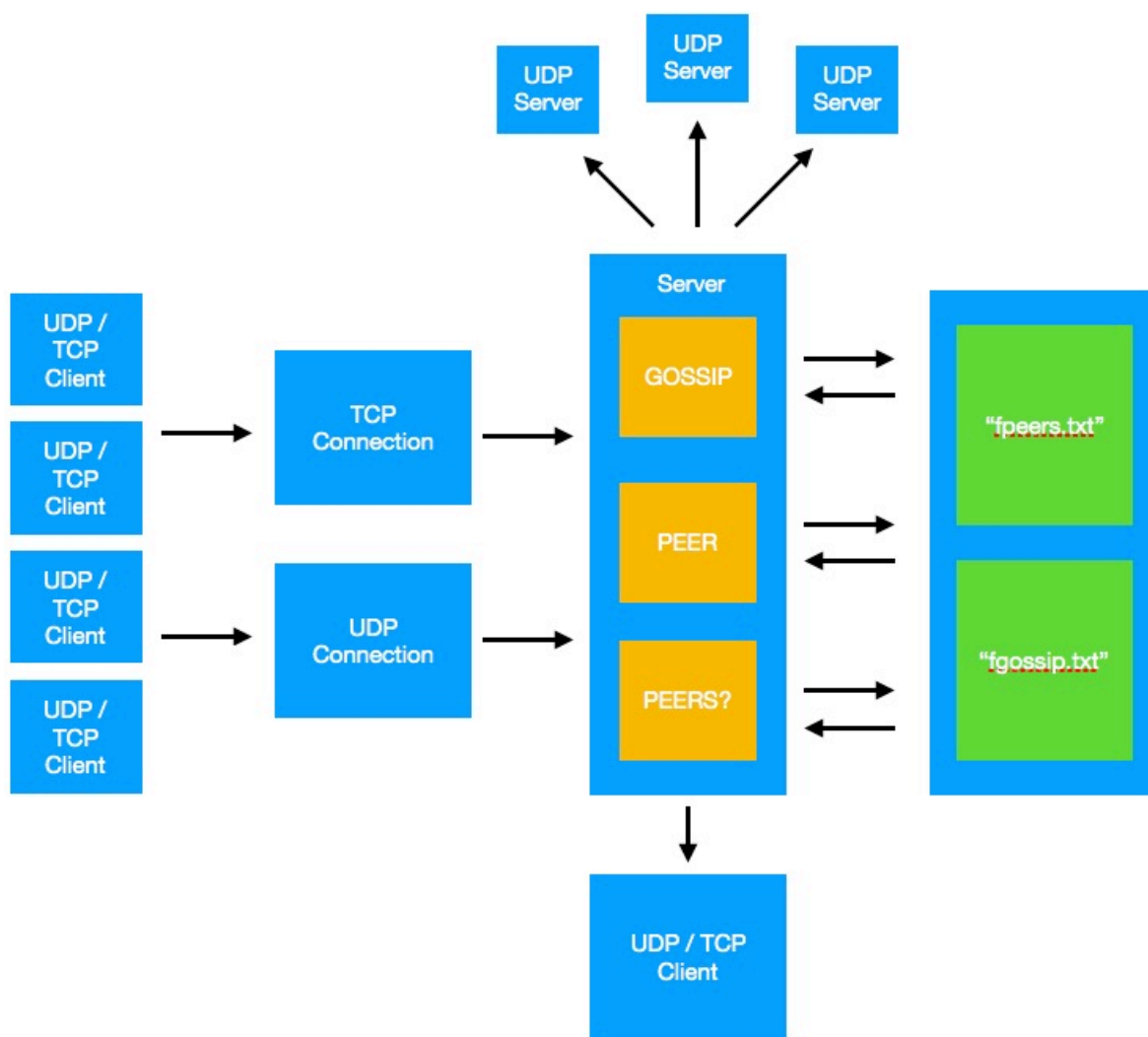
- C language : Programming language used to write the server.
- Java language : Programming language used to write the client
- Vim : code editor
- Xcode : IDE used to write and compile C language.
- NotePad++ : Code editor.
- Github : Online service for collaboration in the project.

- Bash scripting : Computer language used to automate compilation, execution, and testing
- Sock : Networking tool to test the iterative server.

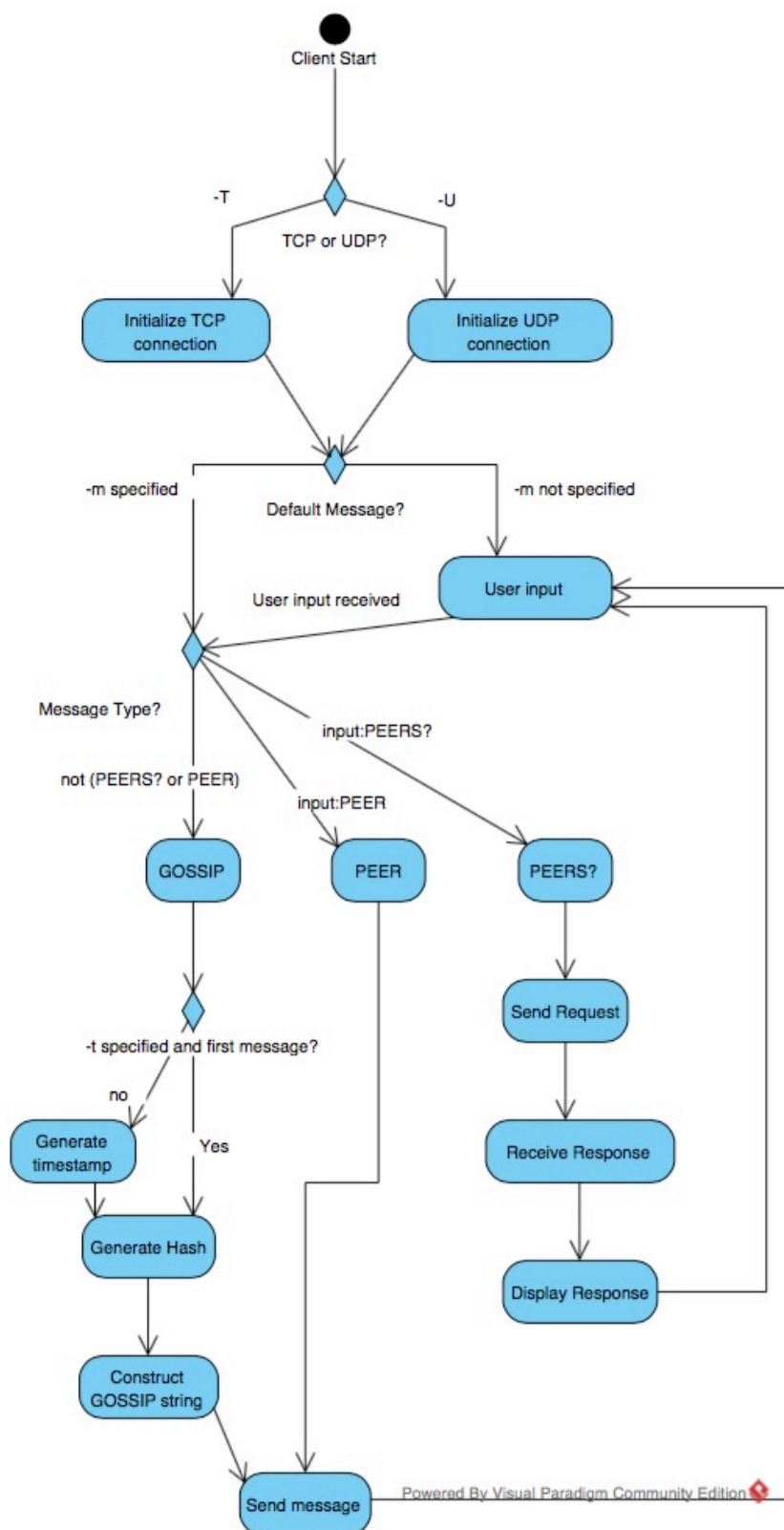
Architecture

Overall Architecture

Milestone 2



Milestone 3

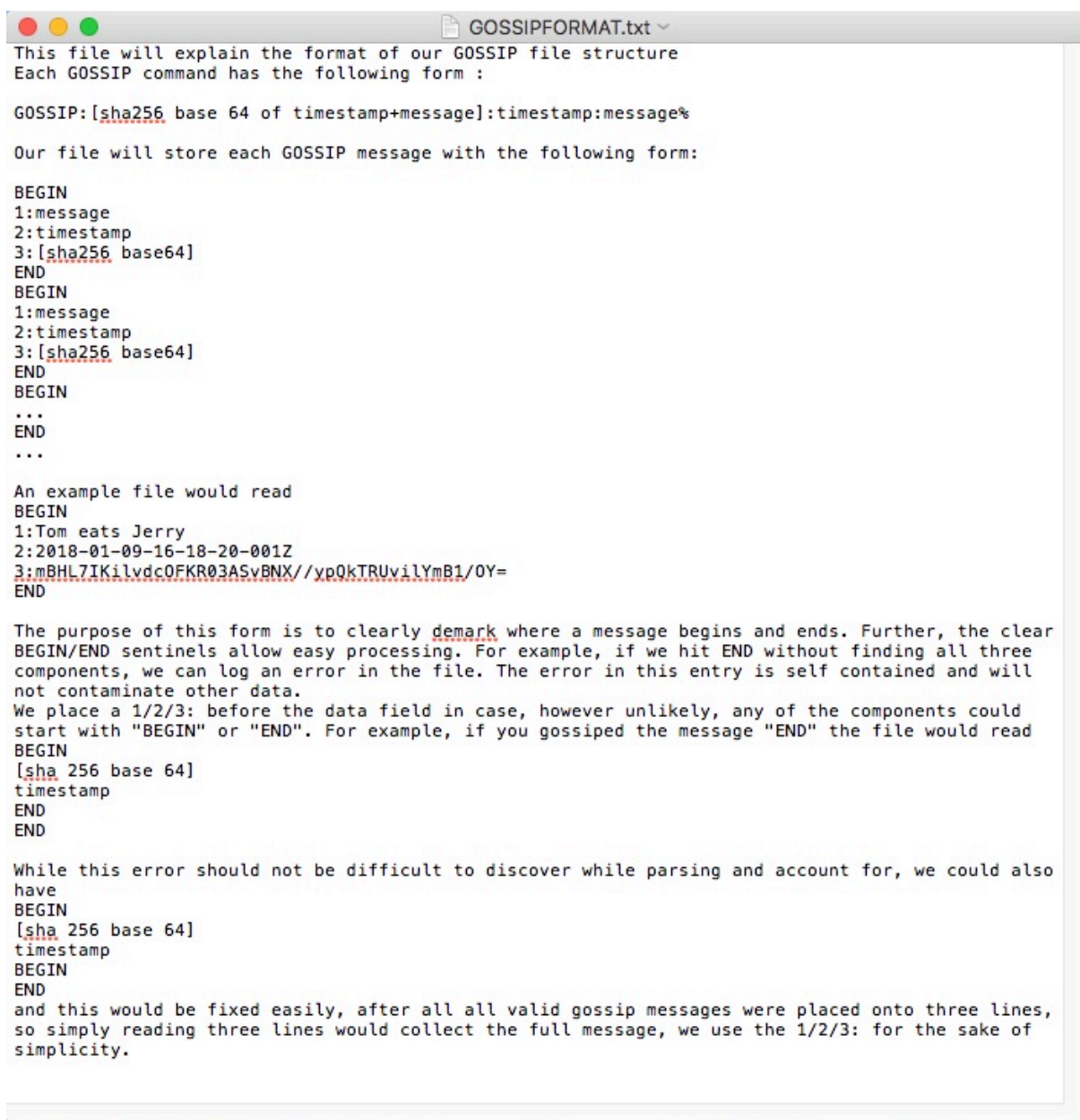


File Structure

Milestone 2 / 3

Files were used as a method of storing data for the server. Two files were sufficient for organizing all the data that had to be stored. One file was used to store gossip messages and the other file was used to store peers.

“FGOSSIP.TXT” FILE STRUCTURE



```

GOSSIPFORMAT.txt
This file will explain the format of our GOSSIP file structure
Each GOSSIP command has the following form :

GOSSIP:[sha256 base 64 of timestamp+message]:timestamp:message%

Our file will store each GOSSIP message with the following form:

BEGIN
1:message
2:timestamp
3:[sha256 base64]
END
BEGIN
1:message
2:timestamp
3:[sha256 base64]
END
BEGIN
...
END
...

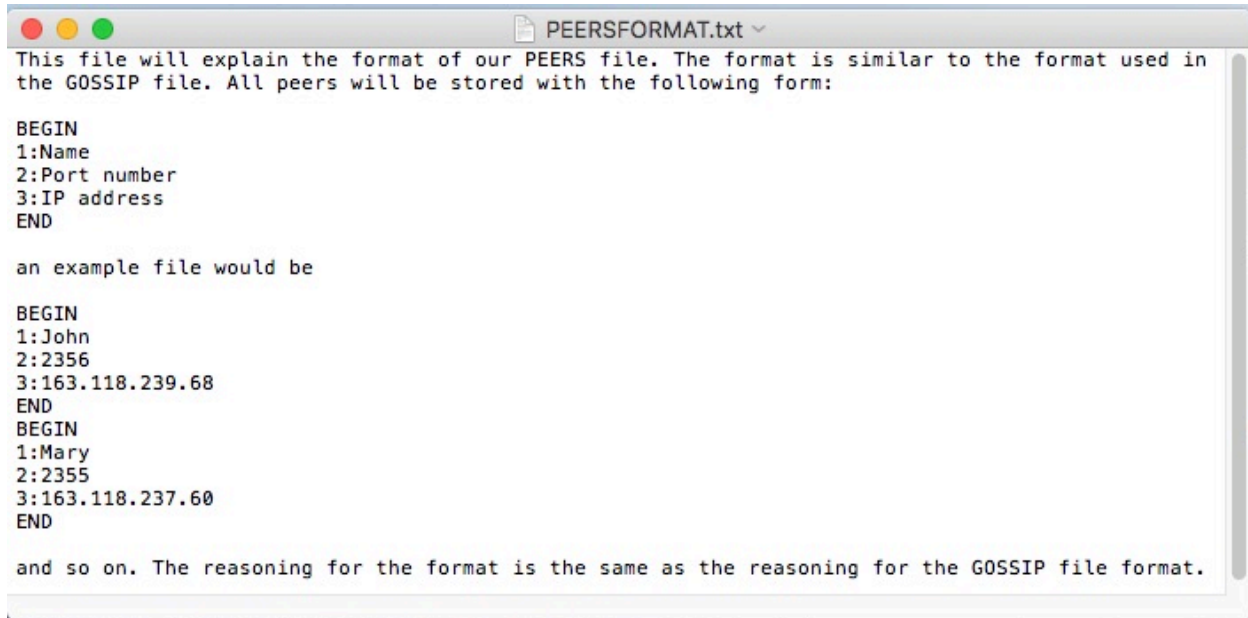
An example file would read
BEGIN
1:Tom eats Jerry
2:2018-01-09-16-18-20-001Z
3:mBHL7IKilydc0FKR03ASvBNX//ypQkTRUvilymB1/OY=
END

The purpose of this form is to clearly demark where a message begins and ends. Further, the clear
BEGIN/END sentinels allow easy processing. For example, if we hit END without finding all three
components, we can log an error in the file. The error in this entry is self contained and will
not contaminate other data.
We place a 1/2/3: before the data field in case, however unlikely, any of the components could
start with "BEGIN" or "END". For example, if you gossiped the message "END" the file would read
BEGIN
[sha 256 base 64]
timestamp
END
END

While this error should not be difficult to discover while parsing and account for, we could also
have
BEGIN
[sha 256 base 64]
timestamp
BEGIN
END
and this would be fixed easily, after all all valid gossip messages were placed onto three lines,
so simply reading three lines would collect the full message, we use the 1/2/3: for the sake of
simplicity.

```

“PEERS.TXT” FILE STRUCTURE



```

This file will explain the format of our PEERS file. The format is similar to the format used in
the GOSSIP file. All peers will be stored with the following form:

BEGIN
1:Name
2:Port number
3:IP address
END

an example file would be

BEGIN
1:John
2:2356
3:163.118.239.68
END
BEGIN
1:Mary
2:2355
3:163.118.237.60
END

and so on. The reasoning for the format is the same as the reasoning for the GOSSIP file format.

```

Program Structure

Milestone 2

Section A: Implementation of command line options

Implementation of command line options is found inside of main method in server.c at lines 72-83. Options processed are -p for port and -d for path. Option -p is valid for any port from 0 to 65535. Using an invalid port will cause errors further in the program. Option -d specifies the directory for our PEER and GOSSIP files. The directory is hardcoded inside run.sh, which specifies “testFolder” as the directory (testFolder will be created if it does not already exist).

Section B: Initialization of TCP and UDP servers

(all line numbers in this section refer to server.c unless otherwise mentioned)

Initialization of TCP and UDP servers is found inside of main in server.c at lines 86-95 for the TCP server and at lines 98-105 for the UDP server. All connections are handled by a polling mechanism inside of main in server.c, lines 108-140. The TCP server will spawn a new process (see: `tcpConnection(int, char *)` in next section) to handle the client (lines 122-135). The UDP server will not spawn a new process and will directly handle the connection within the same process (see: `udpConnection(int, sockaddr_in, char *)` in next section) in lines 137-139. A failure of either server to bind is not a fatal error and a warning will be written to STDERR informing that BIND failed.

Section C: Where messages are read from sockets

(all line numbers in this section refer to server.c unless otherwise mentioned)

Whenever the server receives a message the polling mechanism will trigger the proper connection function as defined in Section B.

tcpConnection(int, char *): implemented in lines 264-326, `tcpConnection` is run by a child process to handle the TCP sockets interactions. A child process is used due to the stream nature of TCP. In `tcpConnection` we must account for message fragmentation and message concatenation. To accomplish this we define one buffer, here after referred to as `BUF_LARGE`, to hold the total read messages, and one buffer, here after referred to as `BUF_SMALL`, to hold each individual read from the socket. Each read is placed into `BUF_SMALL` and then appended(see `bufAppend(char *, char *, int, int)`) to `BUF_LARGE`. This is to handle message fragmentation(lines 274-275). The result of this operation is that a socket receiving:

Read 1: PE

Read 2: ER:Name:port:IP

Read 3:%

will assemble into BUF_LARGE: PEER:Name:port:IP%. The appending operation also allows for message concatenation, a TCP socket receiving:

Read 1: PEER:Name:port:

Read 2: IP%PEERS?

Will result in BUF_LARGE containing PEER:Name:port:IP%PEERS?. After each command is executed it is cleared from the buffer(see Section D, clearBuffer(char *, int, int)).

udpConnection(int, sockaddr_in, char *): implemented in lines 399-416, the UDP connection does not spawn a child process. This is because each UDP packet is the full set of data, there does not exist a stream to account for unlike with TCP. The message is read in line 405. Again, because UDP is not a stream there does not exist any need to handle message fragmentation or concatenation. It is assumed that each UDP packet contains one and only one command.

Section D: Where messages are parsed

(all line numbers in this section refer to server.c unless otherwise mentioned)

Before messages are pared, the server does a validity test before sending the command into the appropriate methode. That is, all messages sent to the server obey the form

GOSSIP:[sha]:[time]:[message]%

PEER:[name]:PORT=[port]:IP=[ip]%

PEERS?\n

Each message must be parsed in its appropriate protocol handler to determine which helper function to call. After input validation the program determines which function to call. The function `isValidForm(char *)`, described in more detail in the appendix, affirms the message is a valid command string and passes it to the appropriate function `GOSSIP()`, `PEER()` or `PEERS()`.

In `tcpConnection()` this is in lines: 295, 308, 315. In `udpConnection` this is in lines: 409, 411, 413

Both `tcpConnection()` and `udpConnection()` call the same helper functions which handle the message parsing and execution.

GOSSIP(char * char *): first allocates buffers for each of the three GOSSIP string components(`[sha]`, `[time]`,`[message]`)(lines 432-437). Then we place the parts of the GOSSIP string into the appropriate buffers in lines 440-448 knowing that ‘:’ divides them. The actual execution of GOSSIP is described in the methods section.

PEER(char *, char *): first allocates room for each of the three PEER string components(`[name]`,`[port]`,`[ip]`)(lines 671-677). The parsing of the message occurs in lines 686-696, recognizing it follows the same delimiters as GOSSIP with only a need to account for

PORT=, IP= in the last parts of the command. The actual execution of PEER is described in the methods section.

PEERS(int, sockaddr_in, char *, int): does not parse any string, it only reads from our PEER file and writes it to the querying client. The execution of PEERS is described in the methods section.

Section E: Where messages are saved in database

(all line numbers in this section refer to server.c unless otherwise mentioned)

Instead of using a sql server we use fpeers.txt and fgossip.txt (file structure is described above).

Only GOSSIP and PEER strings must be saved, so this section will only discuss these two.

GOSSIP:

All GOSSIP strings are composed of [sha], [time], and [message]. Each GOSSIP string is appended to fgossip.txt in lines 454-460, such that the components are saved in the file in [message], [time], [sha] order.

PEER:

All PEER strings are composed of [name], [port], and [ip]. Each PEER string is appended to fpeers.txt in lines 702-708, such that the components are saved in the file in [name], [port], [ip] order.

Section F: Where messages are forwarded further

(all line numbers in this section refer to server.c unless otherwise mentioned)

Only GOSSIP strings are forwarded further, so this section will only discuss GOSSIP strings.

For a message to be forwarded further GOSSIP must connect to each known peer and send the GOSSIP string. This is handled in lines 464-467 and the helper function `broadcastToPeers(char *, int, char *)`. Lines 464-467 are a loop to execute `broadcastToPeers()` for each peer.

Milestone 3

Section A: Reading user input

(All line numbers referred to are in Client.java)

The client has three functions to fulfill, as outlined in the activity diagram. It must collect user input, send it forward, and receive a response(if appropriate). To do this we first handle reading user input, which is done in lines 228-244. Trivially we collect whatever the user typed, excluding the '\n' character sent by the system, and send it for further processing inside `messageHandler`(lines 79-86). The message is of one of three forms: GOSSIP, PEER or PEERS?.

The message type is determined as follows

PEERS? - Only the string "PEERS?"

PEER... - Any string that begins with "PEER". The validity of the form is confirmed by the server.

GOSSIP - Any string which does not match either of the previous forms.

For a **PEERS?** command we must perform three steps defined in lines 156-159:

Send the request to the server(Implemented in lines 133-140)

Receive the response string from the server(Implemented in lines 165-177)

Display the response to the user(Implemented in lines 179)

We block the user from entering further commands until the server has responded. This was done to prevent odd text buffering on the command line experienced during early testing.

For a **PEER** command we need only perform a single step, to send the string to the server. This is again implemented in lines 148-150.

A GOSSIP command has more work put into it. The steps of a GOSSIP command are to take the message, generate a timestring, generate a hash and send this string off. However, the existence of initial messages and timestamps slightly complicates the process.

Generating a timestring(Lines 113-118) is accomplished with a little trick. The process is as follows:

- 1: Generate a timestamp regardless
- 2: If the initial timestamp exists(non-null), replace the generated timestamp with the initial timestamp
- 3: null out the initial timestamp

The nulling of the initial timestamp is because it can only be used once, further messages are obviously generated at a different time.

Following this, we need to generate a hash. This is handled in lines 124-127. The Base64 hash is simply the SHA256 digest of [timestamp]:[message] encoded into base64.

Finally, we assemble and send off the complete string in lines 105-106, and 134-140.

Section B: Generating the hash

Generation of the hash is handled in lines 124-127. The Base64 hash is simply the SHA256 digest of [timestamp]:[message] encoded into base64.

Section C: Server threads starting point

(All line numbers past this point are in server.c)

Threads are created at line 138-139 with the `pthread_create()` function which calls function `tcpConnection()`, located at line 269. Programs that use threads have synchronization issues because two threads can access the same data at the same time, so in the server we used a pair of semaphores initialized in lines 90-91. The only parts of the program that require synchronization are the parts that deal with accessing the `fpeers` and `fgossip` files. To keep thread safety while maintaining simplicity we only allow one thread to write to the files at a time (Line pairs (459,469), (743,757)) .

Method Description

Milestone 2

(Meaning of variable names are specified in the header every method in server.c)

int isValidForm(char *); takes a string to parse. The function will determine if it fits the form of a GOSSIP, PEER or PEERS command. A string is said to have the form of a GOSSIP command if it obeys the structure GOSSIP:[sha]:[time]:[message]% such that [sha] is 44 characters long, [time] is 24 characters with 6 '-' characters, and [message] is an arbitrary string. A string is said to have the form of a PEER command if it obeys the structure PEER:

[name]:PORT=[port]:IP=[ip]% such that [name] is an arbitrary string, [port] is any number having at most 5 digits, and [ip] obeys the form: [a].[b].[c].[d] where a,b,c,d are any numbers having at most 3 digits. A string is said to have the form of a PEERS command if the string is "PEERS?" exactly. Any string which does not obey one of the above forms is rejected as malformed.

void tcpConnection(int, char*); Discussed in details in SECTION C.

int bufAppend(char*, char*, int, int); A wrapper for strcat that ensures the source buffer will fit into the destination buffer before execution, otherwise it will return -1. This was primarily used during early testing, however the return value is no longer relevant as of submission build.

void clearBuffer(char*, int, int); clearBuffer removes all content up to howFar, shifting down all content past that point. All memory that is shifted down has its original location in the buffer overwritten with '\0' to ensure memory safety and avoid potential errors while processing the buffer in the future.

int removeNewLines(char*); is a function that is used by udpConnection and tcpConnection to remove all newline characters '\n' of a given string. Basically, removeNewLines every time it finds a newline character '\n' shifts to the left the entire content of the string after that '\n'.

Returns the number of '\n' found in the string.

int commandCount(char*); is a function used by tcpConnection to determine how many commands are in a given buffer (char * buf). Tcp connection might concatenate two commands together, so this function is used to make sure that every command in the buffer is executed. Returns the number of commands found in the given buffer.

void udpConnection(int, struct sockaddr_in, char*); Discussed in details in SECTION C.

int GOSSIP(char*, char*); After stripping the relevant parts of the GOSSIP string(already covered) GOSSIP() must determine if a given message is already known. This is done by scanning fgossip.txt for a matching [sha] entry. If this is found we discard the message, otherwise we write it to fgossip.txt and broadcast it to all peers.

int isKnown(char*, char*, char); is a function used by GOSSIP and PEER to determine if a given command or peer does already exist in “fgossip.txt” or “fpeers.txt”. IsKnown opens the file specified in the parameters, scans the entire file and returns the line number where the match was found or -1 if any error occurred during execution (i.e. if a file is not properly closed).

void broadcastToPeersTCP(char*, int, char*); has the same functionality as broadcastToPeersUDP, but uses TCP instead.

void broadcastToPeersUDP(char*, int, char*); is a function use by GOSSIP to send all its peers, using UDP protocol, the gossip that the server received. Inside this function, the server acts as a UDP client and sends a message to the port and ip retired by peerInfo() function. UDP was chosen over TCP as the best alternative for broadcasting because with UDP the server did not had to connect with the servers before sending the message. On the other side, TCP had to connect with the other server before sending the message and when TCP tried to connect with

servers that were not available for the moment, it would hang the server for a few of minutes and block other operations.

int peerInfo(int, char*, char*); is a function used by broadcastToPeersUDP to retrieve the IP and PORT of a specific peer that is found in the peer file. PeerInfo opens the peer file, finds the port and ip of the desired peer, returns the port, and passes ip by reference. If any error occurs during the execution of the function, it returns -1.

int peerNumber(char*); is a function used by GOSSIP to determine the number of peers the peer file. Returns the number of peers in the peer file.

int PEER(char *, char *); is a function used by tcpConnection and udpConnection to handle the PEER command. First, this function parses the peer command into three fields: peer's name, peer's port, and peer's address. Then, checks if the peer that was fed to the function was known or not using isKnown() function. If the peer is known to the server, updateFile() function updates peer's address. If the peer is not known to the server, PEER appends peer's information into the peer file. Returns 0 if the update or appending was successful or -1 if any error occurred.

int updateFile(char*, int, char*, char*); is a function used by PEER to update a peer's ip address. Basically, updateFile takes the line number that has to be changed in the peer file, creates a new file named "output.txt", copies every line of the "fpeers.txt" to the new file until it is at the line that has to be updated, writes the new line in the new file, copies the rest of "fpeers.txt" to "output.txt", deletes the old file ("fpeers.txt"), and renames "output.txt" to "fpeers.txt". Returns 1 if the update was successful or -1 if it was not successful.

int PEERS(int, struct sockaddr_in, char *, int); The goal of peers is to respond to a query for the known peers of this server. The response has the form PEERS|#peers|PEER:

[name]:PORT=[port]:IP=[ip]|... To accomplish this we must collect the number of known peers.

Following that is a formula for determining the size of the message. It is easy to discover by counting how many characters exist inside the file, accounting for additional characters(BEGIN,END,1:,2:,3: = 14 per peer) and the need for an additional (PORT=,IP=, :,;,|) in the response. Parses the file, collecting the relevant information from within each BEGIN/END block and put it into the string. Afterwards the function recognizes if the UDP or TCP server called this, and sends a reply appropriately.

int countDigit(int); is a function used by PEERS to count the numbers of digit in an integer. Returns the number of digits.

char* itoa(int, char*, int); is a function used to convert an integer into a string. Returns the number that is passed in a string form.

void error(char*); is a function used to print to standard error.

void sig_chld(int); is a function that is used to handle signals by child processes.

Milestone 3

public Client(String host, int port, String initialTimestamp, boolean TCP); is the constructor of the class Client that establishes a connection with either UDP or TCP depending on the user input. Host and port are specified by the user and are used to create the sockets.

void messageHandler(String input); is a function used by the main function that feeds the command written by the user in the standard input into one of the three functions that handle commands known by the server.

void GOSSIP(String s); takes the message typed by the user, generates a timestamp that fits the format of the server, generates the hash of the timestamp and the message, and send the command to the server using `sendMessage()` function.

String generateTimestamp(); is a function used by GOSSIP to generate the timestamp in a specific format. Returns a string containing the timestamp.

String generateHash(String message, String timestamp); is a function used by GOSSIP to generate the hash of the message and timestamp provided. Returns a string that contains the hash.

void sendMessage(String s); is a function used by GOSSIP, PEER, and PEERS that sends the command (s) provided through TCP or UDP.

void PEER(String s); is a function that sends the PEER command by calling the `sendMessage()` function.

void PEERS(String s); is a function that sends the PEERS command by calling the `sendMessage()` function, and receives the server response with the `recvResponse()` function.

The server sends a string containing the peers that that server knows about.

void receiveResponse(); is a function that is used by PEERS function to listen to TCP or UDP sockets to retrieve data from the server, and then display the response of the server.

displayResponse(String response); is a function used by `recvResponse()` function to display the response of the server in the standard output (terminal).

User Manual

Compile: To compile the server in terminal use `./compileServer.sh`

To compile the client in terminal use `./compileClient.sh`

Execute: To run the server in terminal use `./runServer.sh <port number>`.

NOTE: RunServer script will also compile the server.

NOTE: Run script will create a folder testFolder (if does not exist) to place “fgossip.txt”
and “fpeers.txt”.

To run the client in terminal use `./runClient.sh <port> <ip> [<message> <timestamp>]`

NOTE: <message> and <timestamp> are optional

NOTE: RunClient script will also compile the client.

Test: To test the server use `./script1.sh` in another terminal.

NOTE: Test script assumes that server is connected in port 12345.

To test the client and the server follow these steps:

1. Open 3 terminals
2. Run `./scriptServer1.sh` in the first terminal
3. Run `./scriptServerPeerWitness.sh` in the second terminal
4. Run `./scriptClientTCP.sh` in the third terminal to connect through TCP
5. [optional] Run `./scriptClientUDP.sh` in the third terminal to connect through UDP

Conclusion

Milestone 2

The work done by the server can be summarized as Read, Store, Broadcast, Reply. The work done by the milestones functionality did not benefit greatly from the reliability of TCP ,and especially with broadcasting the simplicity of UDP made it the more favored protocol.

Development took longer than expected, and this can be attributed to inexperience in the material and a poorly coordinated development process. In future milestones we will define more of the structure of the result from the start rather than evolve into the structure of the submission build.

Milestone 3

Implementing milestone 3 required less work than the server. While the server has much more to do and so this is expected, we also believe that greater familiarity with networking concepts and a thorough planning process helped cut down on development time greatly. In the future we will keep a planning stage before implementing features. We still need to communicate better when implementing details, however we have made significant improvements in this area.

References

<http://www.strudel.org.uk/itoa/>