SDS Group Project Report

Group 7 – Botnet

Cody L, John D, Zak A and Will N | Software Development and Security | Report Date: 24-05-2020

TABLE OF CONTENTS

1. Introduction

1.1 Purpose

Our group project aims to create a proof-of-concept terminal-based bot-net coded in C, targeting the GNU/Linux operating system. It will comprise of master and slave programs, where the client machines (slaves) will be able to receive commands from the server (master), act on those commands, and return the output.

1.2 Scope

For our completed project, server and client source files will be presented. A configurator executable will compile the source files. The server can communicate with the client by sending commands, while the client is able to receive and execute said command from server and report the result back to the server. Additionally, the client will be persistent through system reboot. However, the client will not be able to connect to the server if the server is not running. Once the server is up, the client end will immediately be able to connect to the server. Once we complete the basic functionalities, extra features are to be considered and implemented according to the remaining time and the difficulty of implementing said feature.

1.3 Overview

This report will outline each feature included in our completed project and justify their inclusion. Each member's progress and their assigned tasks will be included in Appendix 5.6 for easier marking. Aspects such as the project's features and constraints will be outlined. Finally, an evaluation and conclusion of the project will be included, along with possible future features of the project.

2. Project Requirements

2.1 Feature requirements

2.1.1 Connectivity

Purpose:

Achieve connectivity between network of clients and the server.

Stimulus/Response Sequence:

Server will listen for incoming connection(s) on a specified port. Client will send request to connect to the remote server at a specified address and port. Server will accept client request and append client information to data structure.

Associated functional requirements:

Function ID	1.2.1.3
Description	Server should listen for incoming connections on specified port
Inputs	Server IP address, port number, file descriptor, socket type, socket protocol
Processing	Server will make a call to socket() to return a FD. This FD is used in a call to bind() which associates the FD with a specified port. Server will poll() FD to 'listen' for incoming connections on specified port.
Outputs	Number of FD with events (incoming connections)

Function ID	1.1.1.1
Description	Client should be able to connect to server
Inputs	Server IP address, port number, file descriptor, socket type
Processing	Client will make a call to socket() to return a FD. This FD is used in a subsequent call to connect(). Additionally, a data structure containing the server IP and port will be passed to connect(), creating a TCP socket connection.
Outputs	Data structure containing server information

Function ID	1.1.2.1
Description	Client should have configurable server IP and port configuration
Inputs	Server IP address, Connection port number, Timeout Value(optional), EXEC_PATH(Terminal)
Processing	Inputs will be passed to server_conf() and client_() functions. They are used to read the client.c and server.c source files into buffer, replace the default configuration with the user inputs and write the buffer into temporary files to be compiled. os.system() function is used to compile the c files and generate the elf files and delete the temporary files.
Outputs	Executable client and server files with the user new configurations

Function ID	1.2.1.1 [Change ID: 1]
Description	Server should be able to accept single/multiple connections
Inputs	FD, FD count
Processing	Server will poll() the FD to see if there is an incoming connection (queued connection limit is set to 20). Server will accept() the queued connection(s) on a first-come-first-served basis, storing the accept()ed client's information in a data structure contained within a linked list.
Outputs	Data structure containing accept()ed client's information, including IP address, port number, host name, associated FD, and pointers to additional structs.

Function ID	1.2.1.4
Description	Server should be able to handle a closed client connection
Inputs	Timeout value, linked list of data structures containing connected client information
Processing	Returned result from recv() is 0 bytes, indicating a close connection. Server makes a call to delete_connection(). This function removes closed connection X from linked list by redirecting pointer of X-1 to X+1. Additionally, whilst interacting with a single client, if select() returns 0, indicating a the timeout threshold has been surpassed, the server will close the connection with the client and delete_connection().
Outputs	Returns int 0

Function ID	1.1.2.2
Description	Client should try to reconnect to server if connection dropped/unavailable
Inputs	Timeout value, server IP address and port number
Processing	During initial call to connect() if a non-zero value is returned, the client will sleep for a variable length and retry connection with server until successful. This functionality is repeated if, during communication, select() returns 0 indicating timeout value has been surpassed on socket.
Output	Data structure containing up-to-date connection information, such as FD, server IP address and port number

2.1.2 Communication

Purpose:

Send commands and receive outputs from clients

Stimulus/Response Sequence:

Server will send a command (specified by the user) to the client using the established network connection. The client will parse the command from the server, execute it and return the output to the server.

Associated functional requirements:

Function ID	2.1.1.1 [Change ID: 1]
Description	Server will parse the user's input
Inputs	User specified command
Processing	The server will check against a set of predefined interface commands. If the input string doesn't match a relevant interface command, it will send the string to all clients in the Connections linked list.
Outputs	Command sent to client

Function ID	2.1.2.1
Description	Client will parse the command from the server
Inputs	Command string sent from server
Processing	The client will check against a list of predefined mode-setting commands. If the input string doesn't match a relevant command, it will send the string to the shell and collect the response.
Outputs	Response from shell

Function ID	2.1.2.2
Description	Client sends response to server
Inputs	Response from shell
Processing	The client will check whether the data returned is empty. If it is, it will send a predefined message notifying the user. If not, it will forward the response to the server.
Outputs	Response to server

Function ID	2.1.2.3 [Change ID: 3]
Description	Server opens SSH-like session on client
Inputs	stdin
Processing	The server will send a predefined mode-setting string to a single client. The client will open a PTY session and set the input and output FDs to its connection with the server. Client will act as a means of forwarding traffic to and from the PTY and server. When another mode-setting string is entered, the session will be destroyed.
Outputs	PTY response

2.2.2 Interface

Purpose:

A means of allowing the user to easily switch between and control different clients

Stimulus/Response Sequence:

When the user inputs a defined string into the terminal, it should allow the user to switch between modes of control, notifying the user of the completion of this switch. It should also allow show how many clients are currently connected and aid the user in its use, should an appropriate command string be entered.

Associated Functional Requirements:

Function ID	3.1.1
Description	Displays a welcome screen on server startup
Inputs	User input to startup the server
Processing	Terminal screen will be cleared, and the program will run a series of printf statement containing the welcome message, options for opening a help menu and option to exit the program.
Outputs	Welcome screen

Function ID	3.1.1.1
Description	Displays a help screen that contains a list of available actions
Inputs	User specified command
Processing	The server will check the user input with a list of predefined interface command, when matched, a print_help_screen function that contains a series of printf statement will run
Outputs	Help menu

Function ID	3.1.2.1
Description	Server displays a list of currently connected clients, including the name, IP address and port number of the client
Inputs	User specified command
Processing	Server will iterate through the stored client information and display them in a specified printf format.
Outputs	Connected clients

Function ID	3.1.2.2
Description	Allows the user to select different mode for sending data
Inputs	User specified command
Processing	After successfully comparing user input with predefined string, it will allow the user to enter commands via fgets. This is placed under a while loop to allow consecutive command input from the user.
Outputs	Enters either all or single mode for sending command

Function ID	3.1.3.1
Description	Displays a help menu containing the options of modes
Inputs	User specified command
Processing	The print_command_help_screen function will run when user input matches the predefined string and runs a series of printf statements.
Outputs	Control console help menu

Function ID	N/A [Change ID: 5]
Description	Ability to output, returned by all clients, to .json file for analysis
Inputs	Client information, returned output from clients, input command
Processing	User will be prompted to output results to .json. Server will check if file with name ‘outputs.json’ exists and act accordingly: File exists – file data will be parsed in to json object and checked for validity. File must contain data in correct .json string format and error message presented if check failed. A second check will be carried out to ensure the file contains matching .json data output by the server by searching for matching key value. Error message returned on failure. If both checks are complete, clients’ returned data, including client information, will be appended to json object and subsequently appended to json file. File doesn’t exists – a file named ‘outputs.json’ will be created. A skeleton json object will be created to hold clients’ information and returned output. Said information and returned output will also be placed in to a json object and appended to skeleton json object. This object will then be written to file.
Outputs	Outputs.json containing returned output from clients’ command execution

2.2.3 Persistence

Purpose:

The client application should open on system startup automatically and remain on the device through reboots.

Stimulus/Response Sequence:

The function will check the .profile file for a line indicating that the application will start on boot. If it is not present, it will write this string.

Function ID	4.1 [Change ID: 4]
Description	Write a startup string to /home/.profile file
Inputs	.profile file descriptor, /home path
Processing	The client will parse the .profile file and determine whether the string already exists in the file. If not, it will locate the executable file's path and construct a string to execute the file. This string will be written to the .profile file. Hides executable file in home directory. CheckELF() function is used to check the existence of the elf in /home. If not, access the process directory in /proc/ and copy exe file to /home.
Outputs	Startup command string, hidden executable in /home

2.2 Non-Functional Requirements

Accessibility:

- User interface should be intuitive and easy to use

Availability:

- The project is based on GNU/Linux operating system; hence the user base and victim base of the project is limited
- The program will run on any Linux distribution with the required packages

Efficiency:

- Time taken when booting a server or a client must be near instantaneous
- Time taken for the client to connect to the server needs to be near instantaneous.
- Any interface between a user and the program shall have a maximum response time of two seconds
- Server needs to be able to receive client execution result in under 120 seconds, or a user set value

Integrity:

- Client information displayed in the .json file needs to represent the correct connection object in the connections data structure
- Interface prompt should not be blocked or captured by another process

Reliability:

- Before server sends data to the clients, the server shall confirm that the receiving client is in a ready state prior to the start of transmission.
- The client needs to indefinitely try for a server connection during an unexpected server timeout event.
- Persistence feature needs to check for the existence of the client executable in the home directory.

Usability:

- The program shall be easy to use by any individuals with basic understanding of the provided features.
- Language used in the user interface shall be concise and easy to comprehend.

Maintainability:

- Error description shall be displayed when an error or bug appears on the user terminal that allows the user to understand the cause of said error/bug.
- Shared functions should be appropriately commented, so that they are easy to implement

Modifiability:

- The end user will be presented with the source code of the programs and thus will have ability to modify the contents.
- Some features of the code will be separated into shared functions in header files to improve modularity.

2.3 Constraints

- The programs are required to run on GNU/Linux
- The project is coded in the C language
- The project must be C99 compliant
- The client file should be as small as possible while maintaining all functionality
- The client should persist through system restarts
- The testing environment for the project should be in a closed environment
- The server should be operated via the Linux terminal
- The x86_64 architecture should be targeted

3.0 Project Plan

3.1 Task List, Allocation and Time to Complete:

Top-Level Section	Allocated To	Total Completion Time	Dependencies
Connectivity	Cody Lobban	7 Weeks	None
Communication	John Dickson	7 Weeks	1.2.1.2, 1.2.1.3
Interface	Will Ng	2 Weeks	1.X, 2.X
Persistence	Zak Al-Darmaki	4 Weeks	1.X, 2.X

It is anticipated that the final two weeks of the project will be spent testing for, and fixing, bugs.

More detail referring to the time to complete tasks, along with task reference numbers can be found in Appendix 8.3. An actual project completion timeline can be seen in Appendix 8.4.

4.0 Individual contributions

John Dickson

I initially started work on a shared header file “simple_networking.h”, which would provide the basis of connecting clients and servers together. This was intended to provide high-level functions for use elsewhere in the program, such that it was easy for others to “wire it all together”. At this time, I started the basic provision of data structures and functions which could later be used for the mesh networking of clients together. Namely, the *message* and *machine* data structures were to be used by the client when forwarding messages on to the correct client. The use of the header file proved useful in reducing merge conflicts as others could work on aspects of the server.c and client.c files, with functions being called in the same way, but being updated “under the hood”.

After I completed work on this, I reworked the connection handling backend to use linked lists rather than an array of file descriptors. This was done to ensure that the server maintained a reasonable level of responsiveness when handling large numbers of connections. Particularly expensive was the removal of clients; In the array implementation, it required the last item in the array to be copied to the position occupied by the item intended to be deleted, then shrinking the array to match the correct size. With the linked list implementation, all that is needed is the “rewiring” of the list to skip out the item to be deleted,

then a free() call. The use of linked lists required the definition of a *connection* data structure. At this point, I created a “network_structs.h” file with all the necessary functions required to use the linked list.

I then went on to look into creating a more SSH-like session feeling to the single client mode. This was achieved by creating a PTY on the client side and creating a bash subprocess. In order to implement this correctly, I utilised a guide by R. Koucha, as referenced below. A fairly comical moment arose when myself and Cody were bug testing the implementation and couldn’t work out why the program kept crashing when entering the single user mode, only to realise it was showing the bash prompt from the client.

The last section of my work focused on bug fixing. There were several difficult to reproduce bugs that were discovered in our bug testing phase of the project. Cody and I spent several long nights trying to debug and fix these issues in shared Visual Studio instances.

Cody Lobban

Responsible for managing the team, I drafted the system requirements into a to-do list and assigned tasks to members based on their individual preferences, expertise and interests.

The initial task assigned to me was designing and implementing server and client connectivity. Using Beej’s guide to Network Programming (Hall, 2020) as a reference guide, I started by implementing functionality to connect a singular client to a server and produced over-commented duplicate files with verbose explanations of how the server.c and client.c worked for my group mates to study and understand. John used the client and server framework to help build the .h files and improve the implementation.

I continued to build upon the connectivity framework, eventually adding the functionality to enable several clients to connect. This was achieved by implementing an array of file descriptors (fd) and using a poll() function to check for activity. The addition of the array would allow for adding and removing fds

for new and existing client connections but was subsequently replaced by a superior method implemented by John.

After the project reached its first milestone, each member bug tested areas of the code. I tested a range of different functions, including interaction with interface, interaction with a single client and with all clients. This round of bug testing unearthed a range of different issues, many of which both John and I worked together to fix, improving the functionality and robustness of connectivity, interaction and error handling.

Once all known bugs had been squashed, I worked on formatting returned output from multiple clients. The idea here was to export to a database and display this information on a website, potentially using Elasticsearch and Kibana to realise this idea. However, it became apparent that this would be too difficult for our team to manage. The solution was to create .json files which are formatted by a browser for quick analysis. The addition of .json files allows for future scalability of the project and its application. Using SQL, it would be possible to conduct large scale analysis of infected systems by building formatted tables specific to analytical needs. This was achieved using an external library json-c (Haszlakiewicz, 2012).

John, Zak and I carried out a final round of bug testing, fixing all bugs found. We did uncover an intermittent bug relating to malloc, but this bug only occurred thrice and was elusive in reproduction, meaning we are unsure what caused it and how to fix it.

Will Ng

Initially, I self-nominated to take up the documentation job as I was aware about my lack of coding ability, I was first assigned to record useful information about this group project, such as project ideas, targets, plans, project concerns and problems, during this task many aspects of the project were discussed, for example, I was very unfamiliar with concept of a bot-net, it was through research and the help from my group mates that I was able to get a grip on what our group is actually working on. After Cody and John completed the basic connectivity and communication framework, I was then assigned to complete a

simple text-based user interface for the server side of the project. At first, I was struggling on it because I had no idea where to start, it was through the help and demonstration from Cody that I was able to start coding, it may have been a very small part and a very easy section of code, but it still took me a while to complete due to my unfamiliarity in coding. During this time, I managed to learn and get use to the process of coding, especially, I found myself understanding the “while” loop a lot better as it is a main feature that allows the user operating the server to traverse through different console mode, beforehand I was only thinking about the “for” loop, this demonstrated to me that different loops are useful in different situations.

After I finished my first coding assignment, I was then tasked to add in a simple extra feature that makes the server showing a waring message when the user attempts to exit the program. I utilised the while loop that I just mastered to implement this feature into the code.

At the time of the task, there were two additional deadlines that took place and everyone in our group decided to priorities those tasks. Therefore, progression on this group project was paused for around two weeks. Upon returning, everyone was assigned bug testing tasks, as the basic structure of the project was completed. I was assigned to test the persistence feature of the code implemented by Zak, and the test results were then shared amongst other group members. As time goes on, other members decided to work on new features and additional bug fixes where I was tasked to start working on the final report of the project. First, I created the template of the report and filled in basic information such as the introduction section. Then I also created the original gantt chart, it was later modified by John which allowed it to be presented in a better format. Lastly, my task was to complete the rest of the report, I used Miller and Roxanne’s documentation (2009) to better understand what non-functional requirements are, and later I was able to complete the general report. For the last day before the deadline, after the initial report being reviewed by the rest of the group members, I realised that the report lacked many details that I have missed, with the help from my group members, we were able to complete the final report on time.

Zak Al-Darmaki

After we decided to make the bot-net project, I started reading more on the ways that botnets use to remain persistent even after the infected machines reboot. I chose to work in persistence duo to its importance to better understand the field of virus development and stability. I focused my research on Linux operating system since it is our targeted OS. I found several methods to achieve the persistence in Linux. First, I looked at Cron daemon and rc.local file to do the job. The common difficulty with them was that they both needed super user privileges to allow editing on them. Then, I found the ability of "~/.profile" file which used to build the log-in shell environment for the user. I learned that if you type a command inside it, it will be executed after startup.

Later after Cody and John finished the basic framework, I started to implement the sequence of the required steps to achieve the task in a persistence function in the client source code file. I used popen() to redirect the /home/.profile absolute path to a pipe and read it via char variable. Then, the function will open ".profile" and check if the executing command is written. If not, it will write it at the end of the file. Additionaly, the function stores the client elf in home directory the first time the program is run in a doted misleading name. Then I tested the function by running the server in a machine and the client in different one and restart the client machine. After that, I added function to check the existence of the executable. So, if the target deleted the file, it will copy the source code exe file from /proc/<pid> file system where each running process has a separate directory managing it. this function should be called after exit command from the server.

Lastly, I worked on a python program that will allow the botnet user to easily configure the server IP address, port number, timeout value and terminal path for both client and server files. I decided to use python for the task because its built-in functionalities for file manipulating. The program asks for user configuration in a friendly terminal interface. Then, it validates the inputs such as the IP address and the

port and write new configured client and server in a new temporary file and with the help of OS library it generates executables compiled files.

I also worked alongside with Cody and John on debugging the entire program and log and fix the bugs we found while Will was writing the documentation for the project.

5.0 Specialised libraries

Our client source file requires no linked libraries for compilation. The server source file requires the pthread library and the json-c library. The former is needed due to the multithreaded design of the server, and the latter for outputting the response from multiple clients into a JSON file. pthread is included within the libc6 (Debian) and glibc (Red Hat) packages, and json-c can be downloaded using the libjson-c-dev (Debian) or json-c-devel (Red Hat) packages.

6.0 Changelog

Changelog I.D	Date	User	Description
1	17/04/2020	J Dickson	Move from array to linked list for connection tracking
2	24/04/2020	C Lobban	Temporarily halt development to focus on other module deadlines
3	15/05/2020	J Dickson	Redesign single client mode to implement session-like functionality
4	27/05/2020	Z Al-Darmaki	Improve persistence by checking for binary in home directory
5	29/05/2020	C Lobban	Add JSON output to multiple client mode

7.0 Conclusion and Future work

In conclusion, we achieved the vast majority of functionality originally intended. This includes the basic functionality of the botnet such as the connectivity/communication framework, persistence throughout reboot and a text-based user interface. We also implemented additional features that were not planned initially, such as outputting the response from multiple clients to a JSON file. Bug tests for the project features were also performed alongside feature progression.

The intended feature of firewall evasion was not completed, due to the complexity of this task. Time management through February to the end of April was efficient, however, the progress for the project was stopped from the beginning of May until the second last week of May, due to other module deadlines at the time. In retrospect, if time management was more thoroughly followed, we feel that we could have completed all aspects of the project. This could have avoided the last-minute rush of commits, which increased complexity when patching bugs in the codebase.

In terms of future work, several ideas were floated at the start of the project for inclusion, should time allow. Due to our limited time budget towards the end of the project, some of these were not included. These are listed below:

SSL Encryption: Due to the relative complexity and low-level nature of the single client input mode, high level libraries for including this feature could not be used. This resulted in us being unable to implement this feature by the project deadline. However, given greater research into SSL and future development, it could be included.

Mesh Networking of Clients: Groundwork for this feature is included in the “simple_networking.h” file. This would have aided in firewall evasion, as a single machine could have been used as a staging point for infection, and subsequently control, of other clients. This would be achieved through the routing of commands to machines without an internet connection, by machines which do. Again, due to time constraints we were unable to implement this, but it would be possible in the future.

8.0 Appendices

8.1 Task List References

1. Connectivity: Cody Lobban
    - 1.1 Client
        - 1.1.1 Client Framework
            - 1.1.1.1 Connect to Server – 1 Week
        - 1.1.2 Deployment
            - 1.1.2.1 Allow configuration of master IP/port - 1 Week
    - 1.2 Server
        - 1.2.1 Server Framework
            - 1.2.1.1 Accept Multiple Client Connections – 2 Weeks
            - 1.2.1.2 Accept Singular Client Connection – 2 Weeks
            - 1.2.1.3 Listen for Client Connection – 2 Weeks
            - 1.2.1.4 Close Client Connection – 1 Week
            - 1.2.1.5 Obfuscate Data – 3 Weeks
2. Communication: John Dickson
    - 2.1 Remote Code Execution
        - 2.1.1 Server: Send Command
            - 2.1.1.1 Parse User Input – 1 Week
        - 2.1.2 Client: Execute Command
            - 2.1.2.1 Parse Command from Server – 1 Week
            - 2.1.2.2 Return Output to Server – 2 Weeks
            - 2.1.2.3 Allow for session-like terminal functionality – 3 Weeks
3. Interface: Will Ng
    - 3.1 Server Interface
        - 3.1.1 Welcome Screen
            - 3.1.1.1 Help Menu – 2 Weeks

3.1.2 Analytic Console

3.1.2.1 Connected Clients – 2 Weeks

3.1.2.2 Jobs List – 2 Weeks

3.1.2.3 Help Menu – 2 Weeks

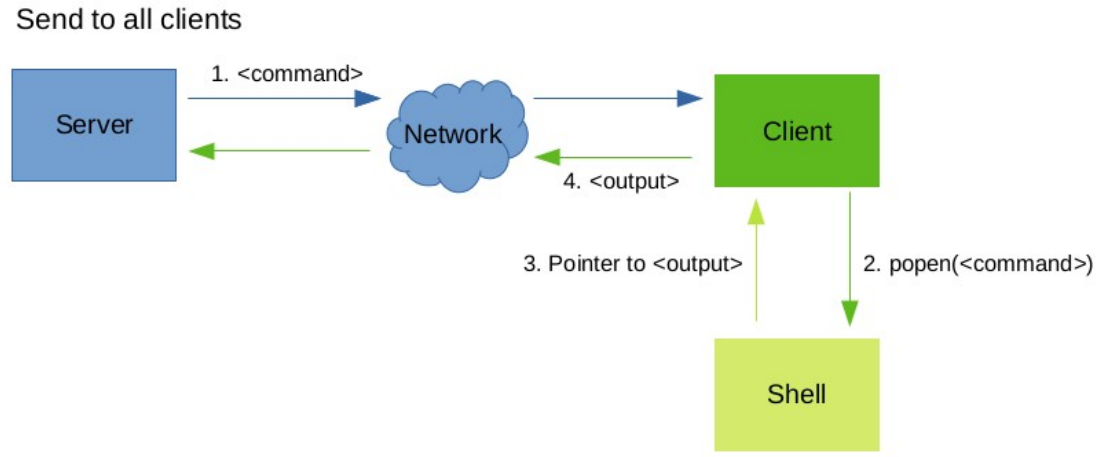
3.1.3 Control Console

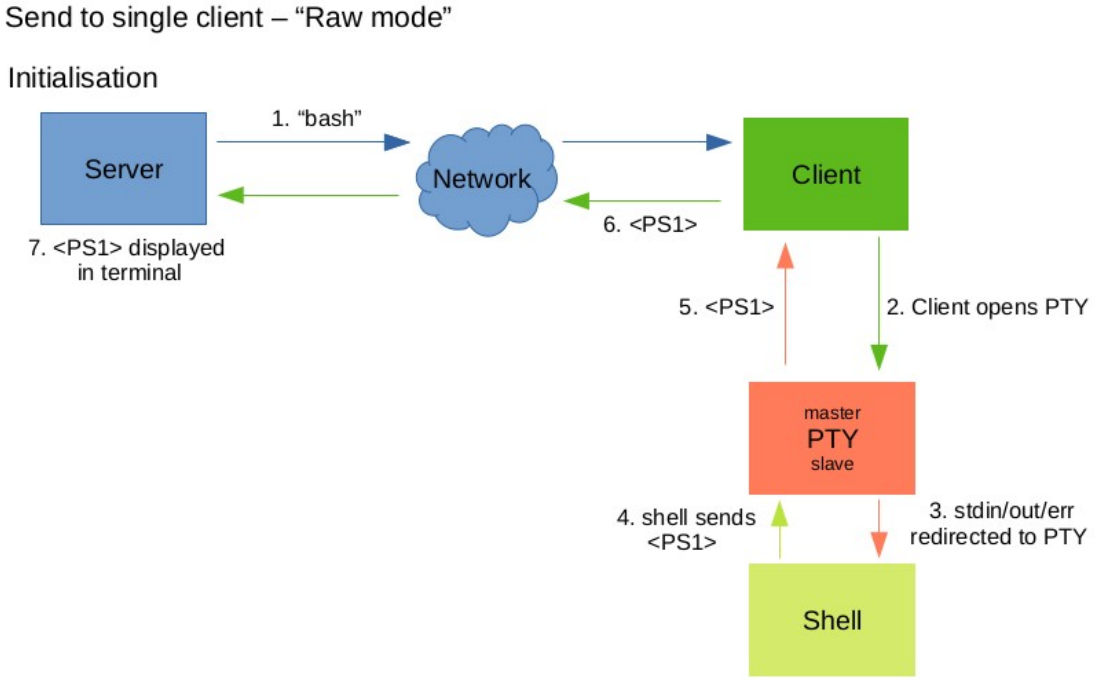
3.1.3.1 Help Menu – 2 Weeks

4. Persistence: Zak Al-Darmaki

4.1 Execute Client on Start-up / Login – 4 Weeks

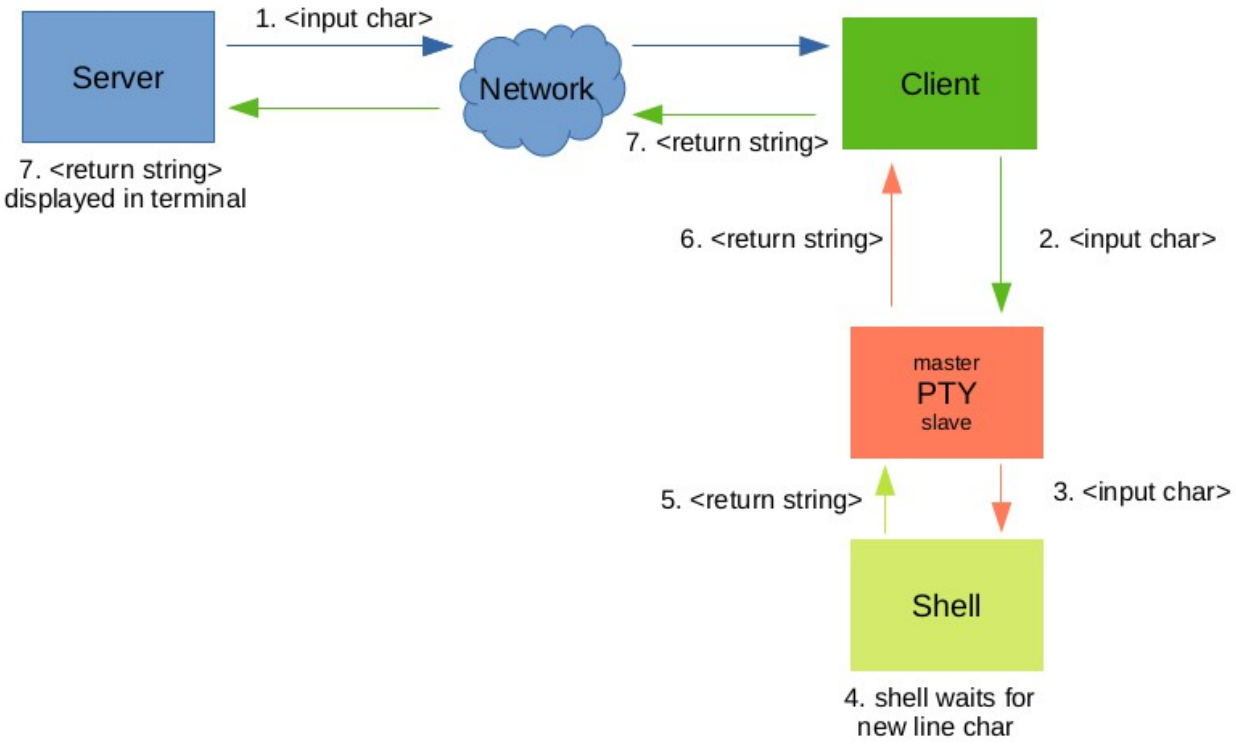
8.2 Communication Block Diagrams





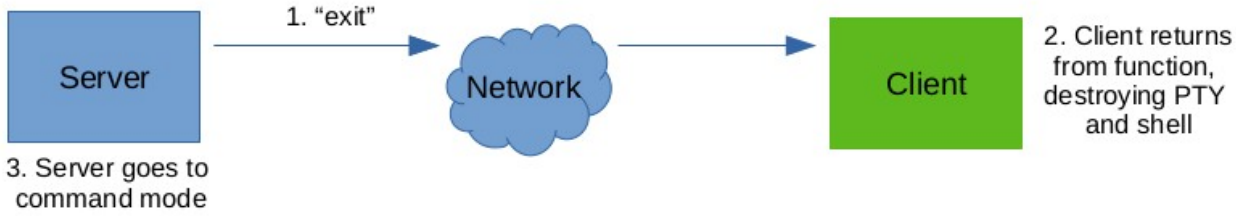
Send to single client – “Raw mode”

In use

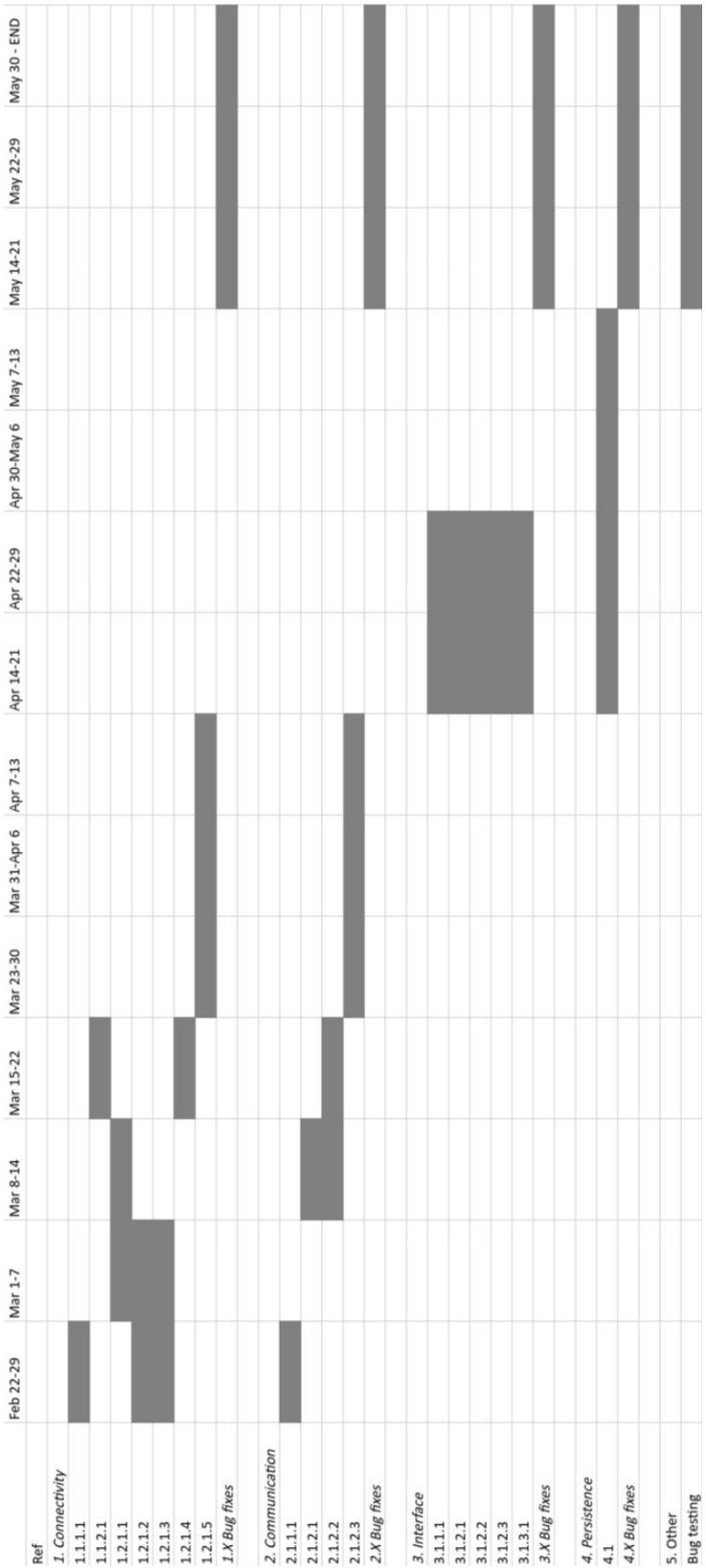


Send to single client – “Raw mode”

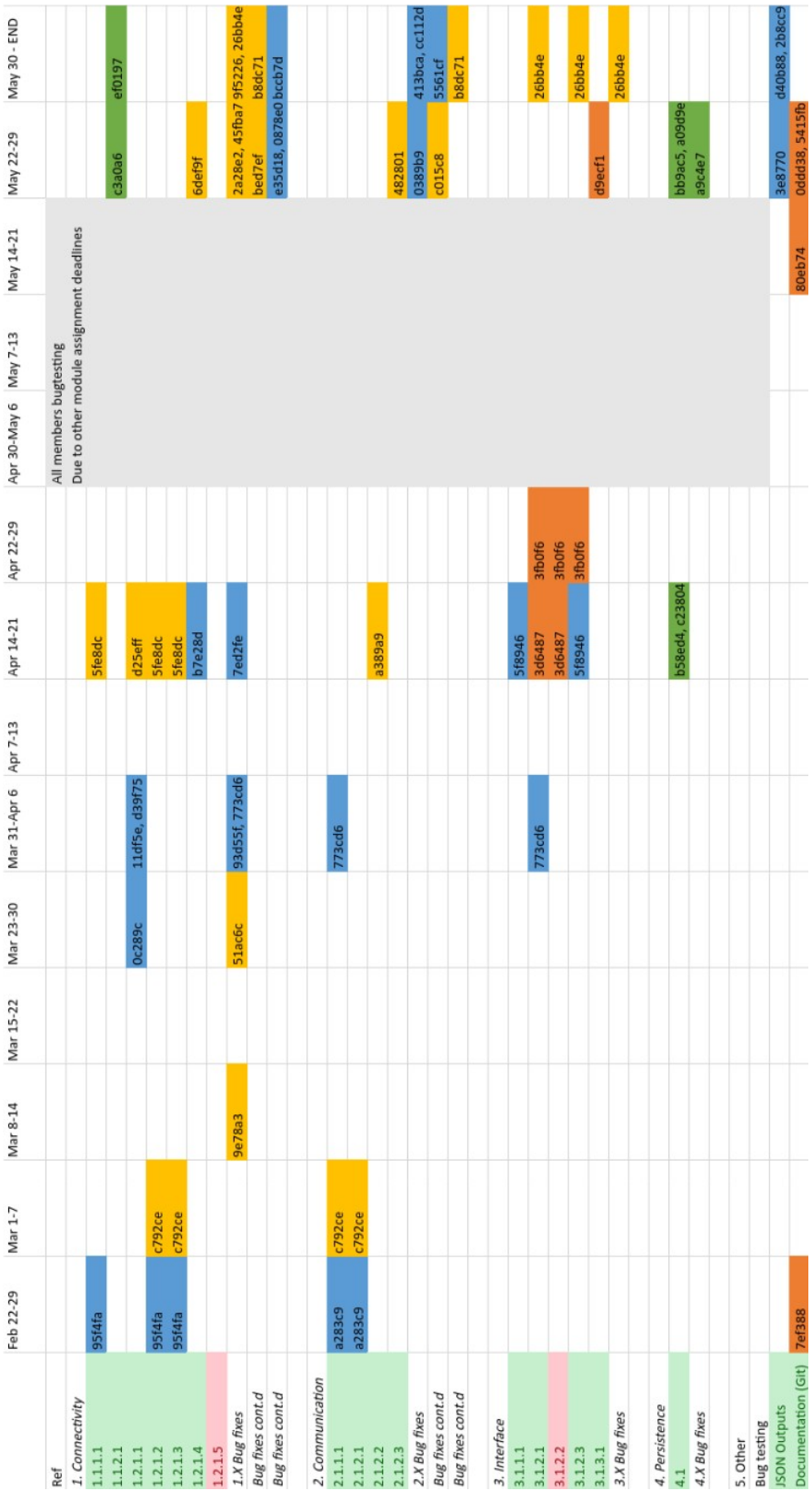
Exit



8.3 Planned Gantt Chart



8.4 Actual Project Timeline Chart – Based on git commits to the nearest week.



The full “SDS Chart.ods” file with both Gantt charts is included in the Git repository.

9.0 REFERENCES

Uses: client.c (Setting up a PTY session)

Koucha,R 2014. Using pseudo-terminals (pty) to control interactive programs. [online] Available from: http://rachid.koucha.free.fr/tech_corner/pty_pdip.html. [Date accessed: 02-03-2020]

Uses: server.c (generating the Json functions)

Haszlakiewicz, E 2012. Json-c Documentation. [online] Available from: https://json-c.github.io/json-c/json-c-0.10/doc/html/index.html [Date accessed: 25-05-2020]

Uses: Get client IP orig. implementation form beej

Hall, B 2019. Beej's Guide to Network Programming Using Internet Sockets. [online] Available from: https://beej.us/guide/bgnet/html/#getpeernameman [Date Accessed: 25-03-2020]

Uses: Setting up connectivity of client and server

Hall, B 2019. Beej's Guide to Network Programming Using Internet Sockets. [online] Available from: https://beej.us/guide/bgnet/html/ [Date Accessed: 01-03-2020]

Uses: Allowed better understanding of non-functional requirements

Miller, Roxanne, E. 2009. The Quest For Software Requirements, MavenMark Books [online] Available from: https://requirementsquest.com/wp-content/uploads/2017/01/Nonfunctional-Requirement-EXAMPLES.pdf [Date Accessed: 29-05-2020]