Assignment 1 – Milestone 2

2803ICT Systems and Distributed Computing

School of ICT, Griffith University

Jessy Barber – s5138877

Contents

1. Problem Statement

The goal of assignment 1 milestone 2 is to implement a simple server to client TCP connection that communicates based on message protocols defined to play the numbers game. The assignment requires the implementation of two main programs, the game server which acts as the game manager, and the game client which is used by players to communicate with the game server as they play the numbers game. This remote execution system takes user input in combination with the specified protocols to send and receive information between the game server and the game client.

2. User Requirements

The following outlines the user requirements for the program:

1. The user joins the game server by running the game client program with the arguments <Game Type> <Server Name> and <Port Number>. E.g. game_client numbers mypc 4444.
2. After the game has started, the user can enter any input which will be handled by the strict message protocols between the server and client.
3. The user shall be able to quit the game by inputting ‘quit’.
4. The user shall be able to have their turn by inputting a number between 1 and 9.

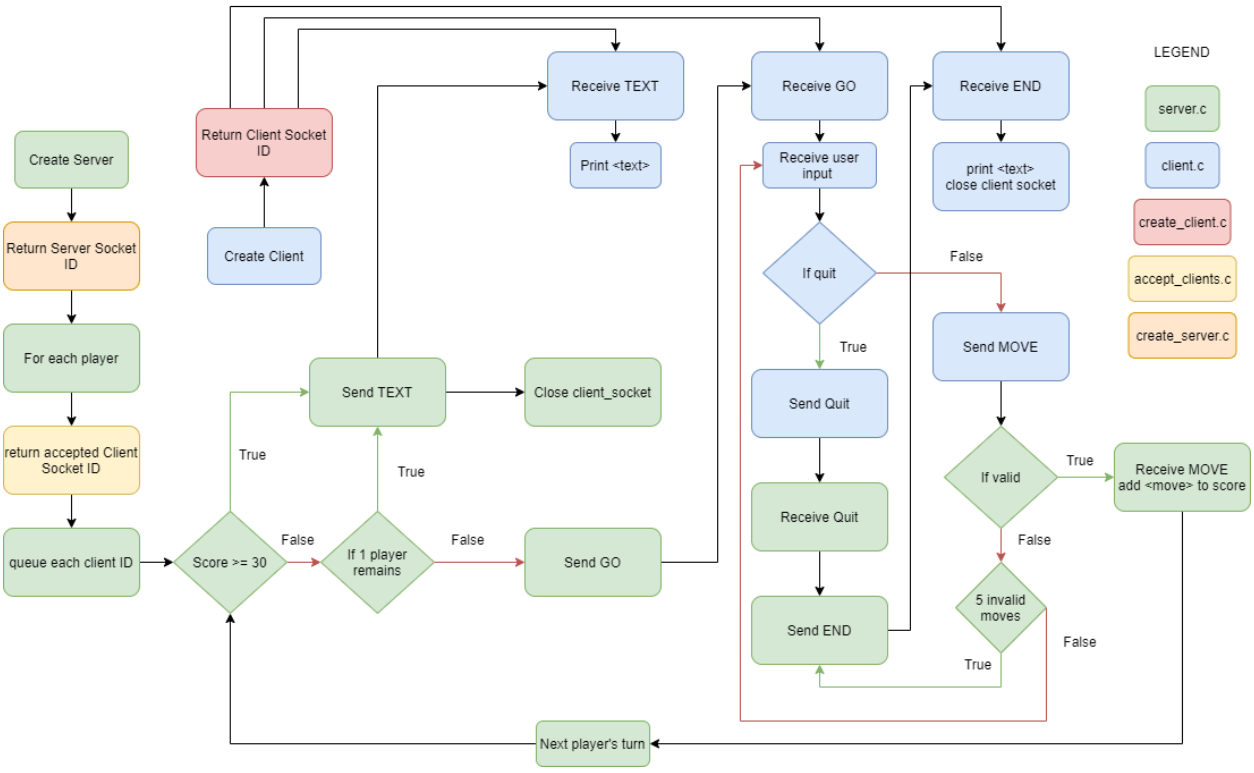
3. Software Requirements

The following outlines the software requirements for the program:

1. The Game Server shall listen for join requests and open a separate communication channel for each client. When a new player joins, the Game Server send a text message saying, “welcome to the game”. The Game Server shall wait for required number of players to join as defined by the <Game Arguments> argument before starting the game.
2. The Game Server iterates amongst the players in their joining order.
3. The Game Server sends/receives messages from the client based on the game and handles all potential user inputs. If a problem prevents the game from continuing, then the client should close all of its resources and terminate itself. A player who performs 5 consecutive game errors is disconnected from the game. The Game Server sends messages to the clients at the end of the game announcing the winners and losers. All client sockets and the server socket should now be closed.
4. The Game Server ends the game if the number of players falls below the minimum required players of 2. The sum of all previously chosen numbers by the players is stored on the server and displayed to each player prior to their turn. The Game Server ends the game when the sum of the chosen numbers equals or exceeds 30.

4. Software Design

High Level Design – Logical Block Diagram



List of all functions in the software:

create_server (appears: create_server.c & server_header.h & server.c):

- The create server function creates the server by creating sockets, binding these sockets and listening and returning server socket information.
- The create server function takes an int port_number and int player_backlog as its parameters. The port number refers to the port specified by the user when running the server program and the player backlog refers to the maximum number of clients the server can facilitate.
- The create server function is called once at the start of main in server.c and initializes the server on the specified port number.
- The function returns an int which is the ID of the successfully created server_socket.

accept_clients (appears: accept_clients.c & server_header.h & server.c):

- The accept clients function accepts incoming connection requests from clients and stores their client IDs. The welcome message is defined in this function and sent to all accepted clients upon successful connection.
- The accept clients function takes an int server_socket, which is the ID of the server socket created in the create server function as its parameter.
- The accept clients function is called in a while loop at the start of the main in server.c for each number of players entered as an argument when running the server program.
- The function returns an int which is the ID of each client that successfully connects to the server.

server.c:

- The server function facilitates the driver code for the server to operate and calls the accept clients and create server functions. These operations include sending and receiving message protocols to each client, managing game mechanics such as detecting when a player has won the game, when a player is kicked for consecutive invalid moves or timeout, keeping track of the current score, and managing the queue of player turns.
- The server function takes command line arguments int argc and char* argv[] as its parameters. These command line argument specify the port number, game name and number of players.
- The function returns 0 when successfully completed.

queue.c (appears: server.c):

- The queue.c file contains the functions create_queue, is_full, is_empty, enqueue and dequeue. These functions make up what is a FIFO queue used to dictate the order of player's turns.

- create_queue takes an unsigned int capacity as its parameter which defined as the number of players in the game. is_full, is_empty and dequeue take the name of the queue as their parameters. Enqueue takes the name of the queue and int item as its parameters, where int item represents the client ID that will be stored in the queue.
- The enqueue and dequeue functions directly change the values that are stored inside the queue when called in server.c. Enqueue takes a client ID and appends it to the back of the queue if the queue is not full. Dequeue takes a client ID and removes it from the front of the queue if the queue is not empty.
- The is_full and is_empty functions return a 1 if they are full or empty. The dequeue function returns the client ID that was removed from the front of the queue.

create_client.c (appears: client.c & client_header.h):

- The create_client function is responsible for retrieving the host IP from the host name, converting this into the proper IPv4 notation and using this to setup a client socket. The client socket is then created and attempts to connect to the server. Upon successful connection the welcome server message is received and printed to the client's screen.
- create_client takes a char* server_name and int port_number as its parameters. These are both defined as command line arguments when running the client program and represent the name of the server and the server's port number.
- The create_client function is called at the start of the main function in client.c with its input parameters.
- The create_client function returns the ID of the client socket and is stored in client_socket.

client.c:

- The client main function is responsible for sending valid user input to the server and receiving information about this input. The client is responsible for interpreting the received server messages and printing this text to the player's screen. This function is also responsible for closing client socket connections and ending its process.
- The client function takes the command line arguments int argc and char* argv as its parameters. These command line arguments will specify the name of the game, the server name and the server's port number. The server name and server port number are sent to the create_client function.
- The client function returns 0 when successfully completed.

List of all data structures in the software:

- Queue Structure
  - Defined in queue.c and used in server.c to dictate whose turn it is. The queue stores the client IDs in FIFO order.
  - The queue can check whether it is empty or full, and enqueue and dequeue child IDs.

- server.c.

- 1D Arrays
  - Used in server.c and client.c to interpret send / received message protocols.
  - 1D Arrays are defined for the message protocols END, GO, ERROR, MOVE, QUIT and TEXT.
  - server.c, client.c.
- Sockaddr_in structure
  - Used to create client and server sockets.
  - contains useful functions for creating sockets such as sin_family, sin_port and s_addr.
  - create_client.c, accept_clients.c, create_server.c.
- Hostent structure
  - Used to convert the host address to a string in IPv4 dotted-decimal notation
  - Contains useful functions for establishing parameters for Sockaddr functions.
  - create_client.c.
- Pollfd structure
  - Used to detect how much time has passed and signify a user that has timed out.
  - Contains useful functions and symbols for detecting elapsed time.
  - server.c.

Pseudocode:

create_server.c:

If (server sock has been successfully created)

{

- print server successfully created
- setup server using sock_addr structure

If (server socket successfully binded)

{

- print server successfully binded

If (successfully listen for incoming client socket connections)

{

- print waiting for players to join

}

else

- print server socket listen has failed

}

else

- print error binding server socket

}

else

- print error creating server socket

accept_clients.c

- create server welcome message

If(client socket connection successfully accepted by server)

{

- print connection to client has been accepted
- send the welcome message to the client ID
- return the client ID

}

else

{

- print ERROR failed to accept client
- exit

}

server.c:

- create queue with size number of players
- call the create_server function with port number and max players

while (number of players is less than required players to start game)

{

- accept new client connections and store client socket IDs
- enqueue client IDs into queue
- increment number of players

}

while (infinite loop)

{

- Create arrays for incoming and outgoing server messages
- clear the client message buffer

if (player input error counter is 0)

{

- dequeue the current client socket ID

}

If (current score is greater than or equal to 30)

{

- for all players that have not won, send the loser message
- close the loser client socket connections
- dequeue the loser client IDs
- close client socket connections for all loser client IDs

}

If (there is only one player left in the game)

{

- print winner message to remaining client
- close client socket connection for winner client ID

}

- Send the GO message protocol to client and wait for player's move

If (player has timed out)

{

- If the player does not respond in 30 seconds send END message protocol and close their client socket connection.
- Decrease the number of players

}

else

{

- Receive response from client

If(client message protocol is MOVE)

{

If (valid move)

{

- add the player's move to the current total score
- if the total score is equal to or greater than 30 send winner text to winner client ID
- if client has won close their client socket connection
- if client has not won append them to the bottom of the queue

}

else

{

- send error message to client for illegal move
- 5 consecutive illegal moves results in END message protocol being sent to client
- Kick client and disconnect client socket connection.

}

}

If(client message protocol is QUIT)

{

- send END protocol to client ID with quit message
- close the client socket connection
- decrease number of players

}

else

{

- send END protocol to client ID with protocol error message
- close the client socket connection
- decrease number of players

}

}

End while loop

Exit program

create_client.c:

If(Successfully retrieve host IP from the host name)

{

- print valid host name
- convert host address IP to string in IPv4 dotted-decimal notation
- setup client socket using sockaddr_in structure

If(Successfully created client socket)

{

If(successfully connected to server)
{

- receive server welcome message
- print server welcome message
- clear server message buffer
- return the client ID

}

else

{

- print ERROR connection to server failed

}

}

else

{

- print ERROR creation of client socket failed

}

}

else

{

- print ERROR host name was invalid

}

client.c

- create clients by calling create_client function with parameters of server name and port number from command line argument

while(infinite loop)

{

- Initialize 1D arrays for server / client protocol messages
- clear server message buffer
- receive a server message

if(sever message protocol is TEXT)

{

if(server message contains YOU)

{

- print server message to player screen

}

else

{

- print ERROR you have tried to make an illegal move

}

}

else if(server message protocol is Current Sum Is)

{

- print the current sum to the player screen
- clear the server message buffer
- receive next client message

if(client message protocol is GO)

{

- Prompt user for their move

If(client types quit)

{

- send Quit message back to server
- receive END message protocol from server
- print you have quit the game
- close the client socket connection

}

else

{

- send the player's move back to the server with MOVE message protocol

}

}

else if(server message protocol is END)

{

if(server message starts with "Too")

{

- print too many illegal moves message

}

else if(server message starts with "Timeout")

{

- print you have been kicked for timeout

}

else

{

- print you have been kicked due to protocol error

}

- close client socket connection
- exit

}

5. Requirement Acceptance Tests

Software Requirement No	Test	Implemented (Full/Partial/None)	Test Results (Pass/Fail)	Comments (For partial implementation or failed test results)
1	Allow the user to join the game by entering the game name, server name and port number as arguments to the client program.	Full	Pass	User can successfully join the game server.
2	The user can enter any input after the game has started which will be handled by the client-server messaging protocol.	Full	Pass	User can enter any input after the game has started. The result of this will be outlined in the software requirements tests.
3	The user shall be able to quit the game after the game has started by inputting 'quit'.	Full	Pass	User can successfully quit the game by inputting 'quit'
4	The user shall be able to have their turn by entering a digit between 1 and 9	Full	Pass	User can successfully have their turn by inputting a valid number.

6. Detailed Software Testing

No	Test	Expected Results	Actual Results
1.0	Establish Client-Server Connection		
1.1 1.2	- Creation of 2 to a max of 10 game clients attempt to join the Game Server with port number 4444 - Only 1 client joins	For cases 1.1 & 1.2: a. A welcome message is displayed for each client b. The Game Server waits for specified number of players to join For case 1.2 Client instantly wins since less than minimum required players	As expected.
2.0	Player Rotation		
2.1	- Player's turns are iterated through in order of their FIFO position in a queue.	Client is prompted to enter their move in the order that they joined.	As expected.
3.0	Client-Server Communication		
3.1 3.2	- A player wins the game. - A player loses the game.	For cases 3.1 & 3.2: TEXT <text> is sent from the server to a client. A client receiving this message prints the contents of <text> to the screen. For case 3.1: <text> = "You Won!" is printed to screen. For case 3.2: <text> = "You Lose!" is printed to screen.	As expected. Prints <text> to screen.
3.3	- It is a player's turn	GO is sent from the server to a client on their turn. A client receiving this message prompts the user to enter their move.	As expected.
3.4 3.5 3.6 3.7	- quit is entered by a player. - A player has entered 5 consecutive invalid moves. - An undefined protocol error occurs. - Player takes 30 seconds or longer to enter a move	For cases 3.4 – 3.7: END <text> is sent from the server to a client. A client receiving this message should print the contents of <text> to the screen and close the corresponding client socket connection. For case 3.4: QUIT is sent from the client to the server as a response to GO. A server receiving this message should send back an END <text> message where <text> = "You Have Quit the Game" and closes this client socket connection. For case 3.5: <text> = "Too Many Illegal Moves Made" is printed to screen and closes this client socket connection. For case 3.6:	As expected. Prints <text> to screen. Client socket connections successfully closed.

		<text> = “You Have Been Kicked Due to a Protocol Error” is printed to screen and closes this client socket connection. For case 3.7: <text> = “You Have Been Kicked for Timeout” is printed to screen and closes this client socket connection.	
3.8	- User enters 1-4 invalid moves.	ERROR <text> is sent from the server to a client. A client receiving this message should print the contents of <text> to the screen. <text> = “ERROR-> You Have Tried to Make an Illegal Move!” is printed to the screen.	As expected. Prints <text> to screen.
3.9	- User enters a valid move.	MOVE <number> is sent from the client to the server as a response to GO. A server receiving this message should take the string to int conversion of <number> and add this to the current sum.	As expected. Sum is increased.
4.0	Further Game Mechanics		
4.1	- There is only one player left.	If the number of players left is 1, this player wins by default. “You Won!” is printed to screen and the client connection socket is closed.	As expected. “You Won!” is printed to screen. Client socket connections successfully closed.
4.2	- It is a player’s turn.	The sum of previously chosen numbers is stored on the server and printed to the player’s screen prior to their turn.	As expected. Score is printed to screen.
4.3	- Score is equal to or greater than 30.	The win / lose messages are printed as defined in 3.1 & 3.2. All client connection sockets are closed, and the server socket is closed.	As expected. Win / lose messages are printed. Client & server socket connections successfully closed.

7. User Instructions

This program is designed for a Linux environment.

- Extract code from zip file
- In a Linux based terminal, run make -B
- Now run server program
  - ./server <port number> numbers <number of players>
  E.g. for 4 players, ./server 4444 numbers 4
- Now run client program in separate terminals for each player
  - ./client numbers <name of PC> < port number >
  E.g. ./client numbers Jessy 4444
- Program is now running and the game can be played.