Patrick Laverty, Adam Loeckle, Drew Nguyen, Krish Ravi
Dr. Joshi Shital
CS4323
17th February 2022

Group E Mini-Project 1 Progress Report

Git repo: https://github.com/KrishRVH/GroupE

I. Work Distribution

Patrick Laverty

Responsibilities: TCP Server Communication

Create methods on the server side to handle communication for the POSIX Message queue. Need an effective method to listen for any request from the client to handle executions.

Adam Loeckle

Responsibilities: POSIX message queue, server side communication, player creation/structs

Drew Nguyen

Responsibilities:

Client side communication including client-server connection/communication, receiving server/player messages, and sending client messages.

Krish Ravi

Responsibilities:

Krish is responsible for the game logic for single and multiplayer modes, this includes algorithms to test whether the player has entered a valid word, calculating and keeping score, monitoring “passes” and detecting when the game is over, selecting starting player and input.txt file at random etc.

II. Work Completed Thus Far

Patrick Laverty

Completed:

Basic server communication and socket methods. Basic server listening method constructed and socket communication done.

Adam Loeckle

Completed:

Basic POSIX message queue that handles two types of messages (Player struct, character array). Player struct and player creation method.

Drew Nguyen

Completed:

Client socket connections, pseudocode structure for client to server communication, and mapped out basic framework for various functions to receive server to client messages.

Krish Ravi

Completed: Krish has established a framework for the game logic and mapped out how to implement logical elements such as checking word validity and control flow for player turns

III. What is left in the Design?

Patrick Laverty

Remaining Design Elements:

Server to client communication and message handling.

Adam Loeckle

Remaining Design Elements:

Message priority logic, handling different types of messages (structs, ints, chars, etc.), and processing struct data.

Drew Nguyen

Remaining Design Elements:

Implementation of client to server communication.

Krish Ravi

Remaining Design Elements:

Implementation of game logic from pseudo code to C-code, client/server interaction for updating players step by step on game status

IV. Limitations of Code and Remedy

Patrick Laverty

Limitations and Possible Remedy:

Possible issues in handling more than one request at a time. To prevent crashes and overloading, I will implement some sort of queuing process to handle instructions individually.

Adam Loeckle

Limitations and Possible Remedy:

Lack of handling message priorities and processing struct data for player information. The remedy to this is to parse messages based on their respective priority.

Drew Nguyen

Limitations and Possible Remedy:

A limitation could result from the client-side relying on the server to push and pull updates thus resulting in the same issues as the server-side limitations; however, the same remedy applies here as well.

Krish Ravi

Limitations and Possible Remedy:

A possible challenge/limitation will be multiplayer implementation to accurately coordinate two players back and forth with specific time waiting. The remedy for this is to use a structured approach to updating the game and pushing the state to the player.

Code Contributions:

Patrick Laverty

```
#include "Main.h"

void server() {
	int new_socket;
	int new_connection;
	int len;

	struct sockaddr_in server_address;
	struct sockaddr_in cli;

	// Creating Socket
	new_socket = socket(AF_INET, SOCK_STREAM, 0);
	if (new_socket == -1) { printf("Socket creation failed\n"); }
	else { printf("Socket created.\n"); }

	bzero(&server_address, sizeof(server_address)); // Zero out the address space of the
struct


	// Binding socket to the port
	server_address.sin_family = AF_INET;
	server_address.sin_addr.s_addr = htonl(INADDR_ANY); // Convert int to ip byte order
```

```
	server_address.sin_port = htons(BINDING_PORT); // Converting ip port to byte order

	int binder = bind(new_socket, (SA*)&server_address, sizeof(server_address));
	if binder != 0) { printf("Socket bind failed\n"); }
	else { printf("Socket binded successfully\n"); }

	// Socket listener
	int listener = listen(new_socket, 5);
	if (listener != 0) { printf("Listen failed\n"); }
	else { printf("Server listening\n"); }

	// Testing connection
	len = sizeof(cli);
	new_connection = accept(new_socket, (SA*)&cli, &len);
	if (new_connection < 0) { printf("Server accept failed\n"); }
	else { printf("Server accepted the client\n"); }

	//close(new_socket);
}
```

Adam Loeckle

```
#include "Main.h"

// Opens message queue, should only be ran once.
void openMsgQueue()
{
	// Ensures message queue does not already exist and creates a new one
	mq_unlink("/Message_Queue");
	mqd = mq_open("/Message_Queue", O_CREAT | O_RDWR, 0600, NULL);

	if (mqd == -1)
```

```
	{
		perror("mq_open");
		exit(1);
	}
	else
	{
		printf("MQ was opened \n");
	}
}

// Closes Message queue, if using mq_close(mqd) instead of mq_unlink ensure that the
// mqd is already initialized
void closeMsgQueue()
{
	free(buffer);
	mq_unlink("/Message_Queue");
}

// Send message of player struct
void sendPlayerMsg(struct Player player_input)
{
	mq_send(mqd, (const char*)&player_input, sizeof(struct Player), 10);
}

// Send message for game instruction
void sendGameMsg()
{

}

// NOTE: Priority 10 is used for game moves/logic, priority 9 is used for player structs
```

```
// and scoreboard usage

// Since buffer and message size is standard mq_attr.mq_msgsize, the ability to send different
// types is allowed sorted by priority level.

void recieveMsg()
{
	struct mq_attr attr;
	mq_getattr(mqd, &attr);
	buffer = calloc(attr.mq_msgsize, 1);

	unsigned int priority = 0;
	if ((mq_receive(mqd, buffer, attr.mq_msgsize, &priority)) != -1)
	{
		// Player struct messsage. do something
		if (priority == 10)
		{
			struct Player* new_player = (struct Player*)buffer;
			printf("Player: %i, Prio: %i\n", new_player[0].score, priority);
		}
		// Game instruction message. do something
		if (priority == 9)
		{
			mq_receive(mqd, buffer, attr.mq_msgsize, &priority);
			printf("Message: %s, Prio: %i\n", buffer, priority);
		}
	}
	else
	{
		perror("ERROR");
	}
```

```
}

int newPlayer()
{
	int num_players = 3;
	struct Player* players = malloc(sizeof(Player) * num_players);

	printf("Players made");
	return 0;
}
```

Drew Nguyen

```
#include "Main.h"

void client()
{
	int new_socket;
	int new_connection;
	struct sockaddr_in server_address;
	struct sockaddr_in cli;

	new_socket = socket(AF_INET, SOCK_STREAM, 0);
	if (new_socket == -1)
	{
		printf("Socket creation failed \n");
	}
	else
	{
		printf("SOcket created successfully \n");
	}
```

```
        server_address.sin_family = AF_INET;
        server_address.sin_addr.s_addr = inet_addr("127.0.0.1");
        server_address.sin_port = htons(PORT);

        if (connect(new_socket, (SA*)&server_address, sizeof(server_address)) != 0)
        {
                printf("Connection with server failed \n");
        }
        else
        {
                printf("Connected to server \n");
        }

        //close(new_socket);
}

//function for communication between client and server
void /*varnameTBD*/(int new_socket)
{
   char buff[MAX];
   int n;

   //loop for chat
   for(){
      //bzero();
      //accept message from client and copy to buffer
      printf("Enter input: ");
      //print buffer that contains the contents

      n = 0;
      //copy server message from the buffer
```

```
        while((buff[n++] = getchar()) != '\n')
        //send buffer to client
        //write(new_socket, buff, sizeof(buff));
        //bzero(buff, sizeof(buff));


        //in-game logic conditionals for exit
        if(){
            printf("");
            break;
        }

    }

}

//socket creation and verification from client side

void clientToServer()
{

    int new_socket;
    int new_connection;
    struct sockaddr_in serverAddress;
    struct sockaddr_in cli;

    new_socket = socket(AF_INET, SOCK_STREAM, 0);
    if(new_socket == 1){
        printf("socket creation failed");
    }
    else{
```

```
        printf("Socket creation successful");

    }
    bzero();

    serverAddress.sin_family = AF_INET;
    serverAddress.sin_addr.s_addr = inet_addr();
    serverAddress.sin_port = htons(PORT);

    //client socket to server socket
    if(connect(new_socket, (SA*)&serverAddress, sizeof(serverAddress)) != 0){
        printf();

    }
    else{
        printf();
    }

    //function for client-server communication
    /*varnameTBD*/(new_socket);
    close(new_socket);
}

void recieveMsg(int new_socket, char * msg)
{
    memset(/*tbd*/);
    int n = read(new_socket, msg, 3);
    //server conditionals for  reading message from server socket
    error("");
}
```

```
//function for update from server
void receiveUpdate(int new_socket, char /*var name of game state*/)
{
   int player_id = receiveInt(new_socket);
   int /*Player state of board*/(new_socket);
   //update game state
}

//function for receiving player struct
void recievePlayerMsg(){



}


//recieve message for game instruction
void recvGameMsg(){

}




//function for error message
void error(){
   //unfin
   exit(0);
}
```

```
//function for writing from client to server
void sendToServer(int newsocket, int msg){
   //update state
   int n = write(new_socket, &msg, sizeof(int));
   if(n < 0){
      error()
   }
   printf(/*print statement to server*/)

}
```

Krish Ravi

```
//Author: Krish Ravi
/*
Scoreboard logic + Text file from Assignment00
Match starting letter of prospective word to previous word, and then consecutively until end of previous word to check if eligible word
Starting player is random (use random number gen)
if both players pass, new alphabet
if both players pass twice in a row (keep back and forth counter) the game ends
pseudocode to check validity of new word (new)
Precondition: prev is a character array of previous valid word, new is character array prospective new word
for (int i=0; i<sizeof prev;i++)
{
   if (new[0]==prev[i])
   {
      int j = i;
      int k = 0;
      while ((prev[j]!=null) && (new[k]==prev[j])
      {
         j++;
```

```
        k++;
      }
      if (j==sizeof prev)
       word is valid
   }
}
```

Flow of game:

Randomly select input.txt and player1

Display first line to player1, select 1 character and require prev = just that character

word is sent to server, if valid, score updates, player1 gets both player scores from server, then player2 is shown the line from input.txt and so on

During player turn, server sends:

-> input.txt line 1

-> player score

-> other player score

-> Used words

-> prev requirements

During player sending word, server checks

-> received word validity

-> whether word is used

-> assign score as per input.txt (should be based on length)

if invalid

-> penalise for invalid word, request valid entry

if used

-> inform used, penalise

each player gets 4 minutes to enter a word

timer is reset when a word is sent to the server

if player resets the timer 3 times in one turn server considers it a pass

when a new valid word is found, it needs to be added to input.txt and scored accordingly

in singleplayer the server is only allowed to use input.txt words, not dictionary.txt

*/

```
// Game logic, handles recieving/sending messages
void game()
{

}
```