The Chase: A Secure Distributed Application

CS2SNS: Secure Network Services

George Hotten (XXXXXX352) & Izak Fourie (XXXXXX582)

December 23, 2024

Word Count: 1,410

Contents

1 University Declaration

We declare that we have personally prepared this assignment. The work is our own, carried out personally by us unless otherwise stated and has not been generated using paid for assessment writing services or Artificial Intelligence tools unless specified as a clearly stated approved component of the assessment brief. All sources of information, including quotations, are acknowledged by means of the appropriate citations and references. We declare that this work has not gained credit previously for another module at this or another University, save for permitted elements which formed part of an associated proposal linked directly to this submission.

We understand that plagiarism, collusion, copying another student and commissioning (which for the avoidance of doubt includes the use of essay mills and other paid for assessment writing services, as well as unattributed use of work generated by Artificial Intelligence tools) are regarded as offences against the University's Assessment Regulations and may result in formal disciplinary proceedings.

We understand that by submitting this assessment, We declare ourselves fit to be able to undertake the assessment and accept the outcome of the assessment as valid.

All members of the team contributed to all elements of this report.

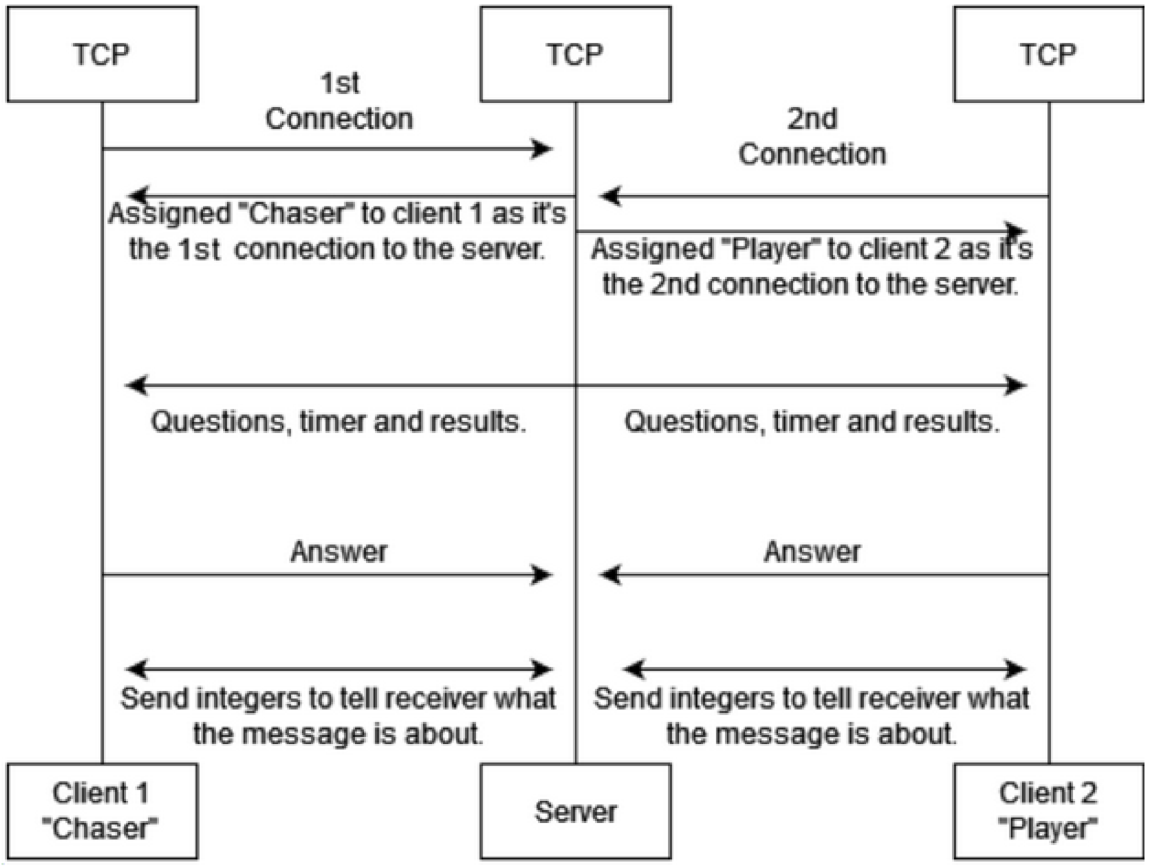
2 Theme and Design

We initially discussed how we would go about making our project, we agreed on using Java for this. Our project is based on the British television show 'The Chase' that first aired back in 2009. The premise of the show is a player goes head-to-head with a chaser, answering multiple choice questions for a chance to win money. To make the game more entertaining and fair, we created the game to be two-players. This removes accusations of the server being biased when giving answers and also allows for friends to compete head to head.

2.1 Architecture

The application uses a key-value-based approach for communication. When sending data, the sender will first send an integer which all parties know the meaning of. Following from the integer, the sender will supply data relevant to the integer. For example, integer 4 is used when a player is sending an answer to the server. After integer 4 is sent, the sender will supply A, B, or C to indicate their answer.

The following diagram shows how the server and the client communicate with each-other:

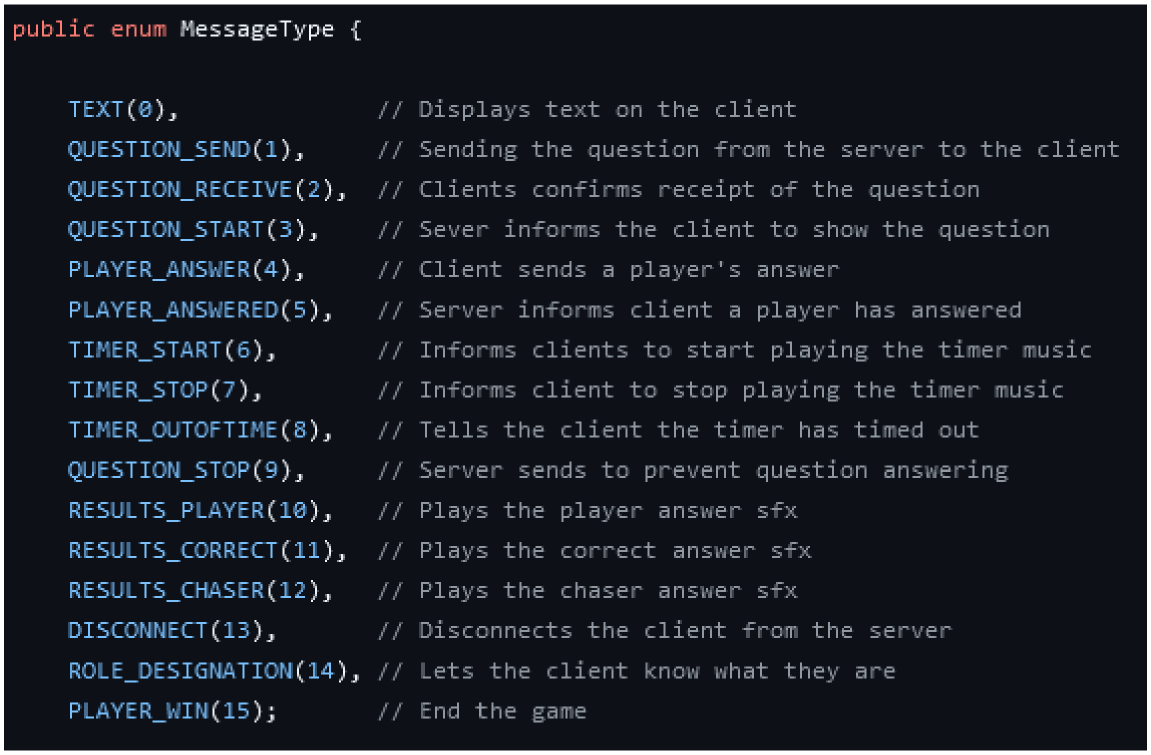

TCP
TCP
TCP
1st Connection
2nd Connection
Assigned "Chaser" to client 1 as it's the 1st connection to the server.
Assigned "Player" to client 2 as it's the 2nd connection to the server.
Questions, timer and results.
Questions, timer and results.
Answer
Answer
Send integers to tell receiver what the message is about.
Send integers to tell receiver what the message is about.
Client 1 "Chaser"
Server
Client 2 "Player"

In this diagram, the client connects to the server using Transmission Control Protocol (TCP). When the first client is connected, they will receive a role based on a "first come, first served" system. The first client will be the chaser, and the second client will be the player. From here, the game will start and data will be sent to the clients containing questions and the possible answers. They submit their answers against a timer and are given back the results.

3 Implementation

Our architecture is the core of our application. Using the before mentioned integer system, these are the different types of messages that can be sent:

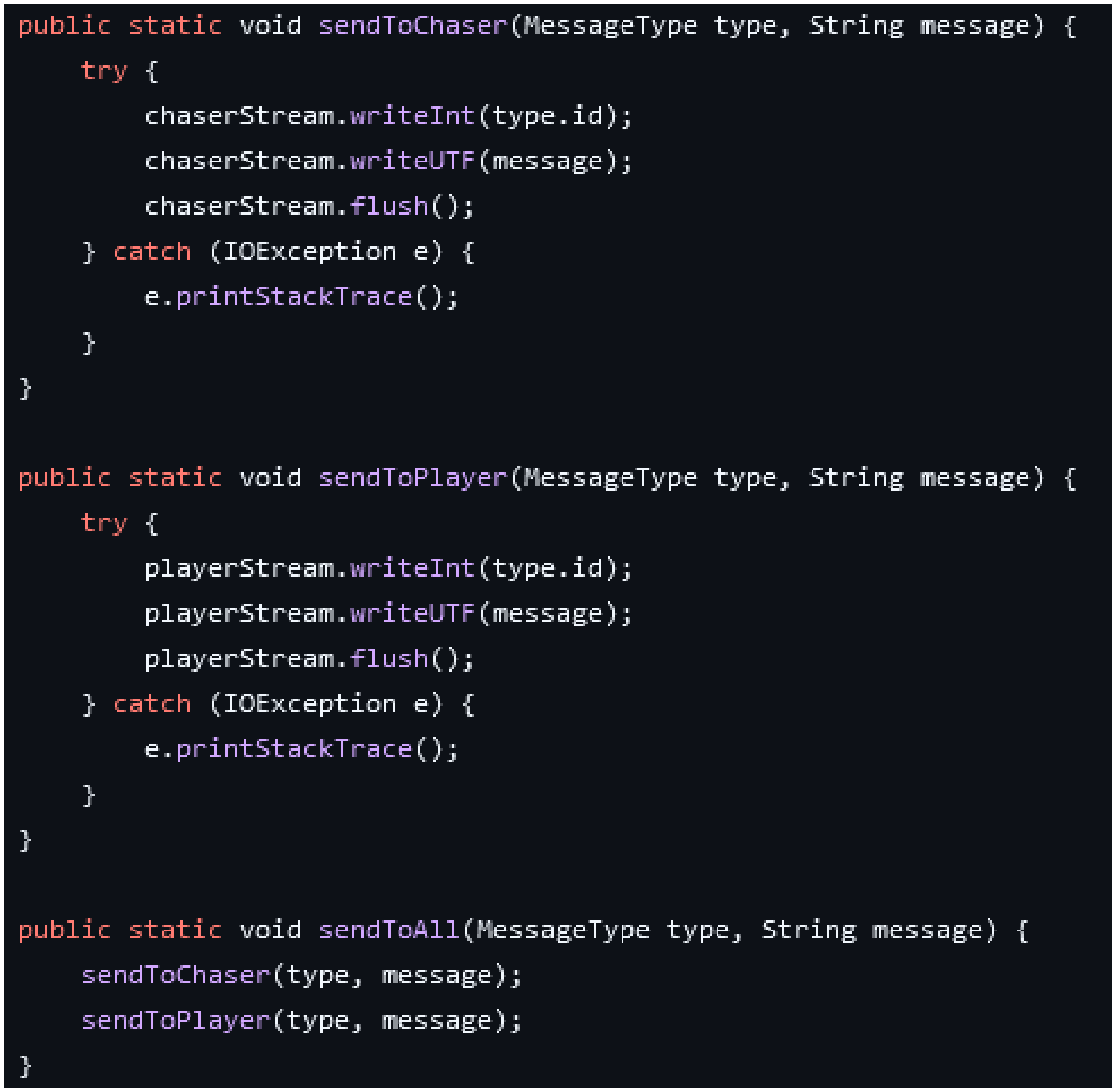
3.1 Message Types



```java
public enum MessageType {

    TEXT(0),                // Displays text on the client
    QUESTION_SEND(1),       // Sending the question from the server to the client
    QUESTION_RECEIVE(2),    // Clients confirms receipt of the question
    QUESTION_START(3),      // Sever informs the client to show the question
    PLAYER_ANSWER(4),       // Client sends a player's answer
    PLAYER_ANSWERED(5),     // Server informs client a player has answered
    TIMER_START(6),         // Informs clients to start playing the timer music
    TIMER_STOP(7),          // Informs client to stop playing the timer music
    TIMER_OUTOFTIME(8),     // Tells the client the timer has timed out
    QUESTION_STOP(9),       // Server sends to prevent question answering
    RESULTS_PLAYER(10),     // Plays the player answer sfx
    RESULTS_CORRECT(11),    // Plays the correct answer sfx
    RESULTS_CHASER(12),     // Plays the chaser answer sfx
    DISCONNECT(13),         // Disconnects the client from the server
    ROLE_DESIGNATION(14),   // Lets the client know what they are
    PLAYER_WIN(15);         // End the game
```

Figure 1: MessageType.java

Each message has a clear comment describing its purpose and when it should be used.

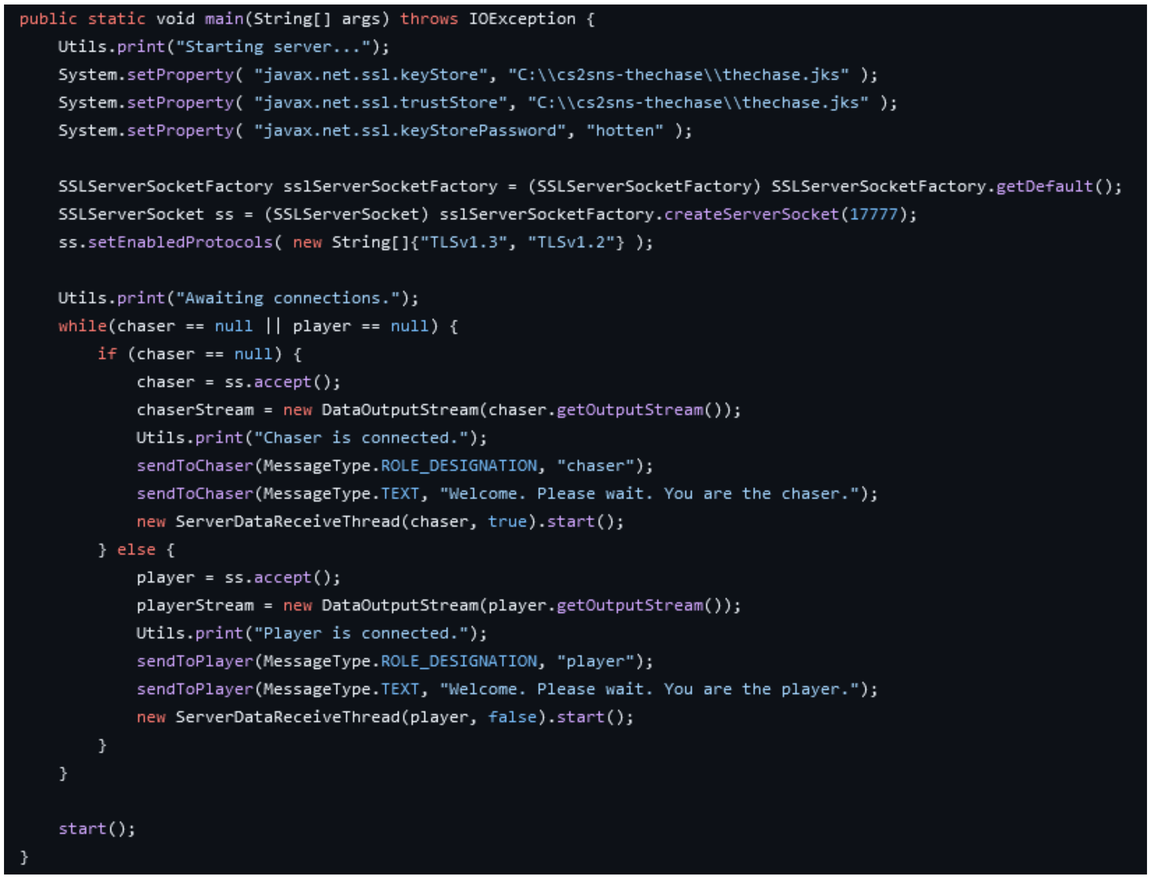


```java
public static void sendToChaser(MessageType type, String message) {
    try {
        chaserStream.writeInt(type.id);
        chaserStream.writeUTF(message);
        chaserStream.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void sendToPlayer(MessageType type, String message) {
    try {
        playerStream.writeInt(type.id);
        playerStream.writeUTF(message);
        playerStream.flush();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

public static void sendToAll(MessageType type, String message) {
    sendToChaser(type, message);
    sendToPlayer(type, message);
}
```

Figure 2: Server.java

To send data to the clients (and for sending data back to the server), the above is used. The message type integer is written to the stream, alongside any string data required for that message type. This data is then "flushed", meaning it is sent to the receiver.

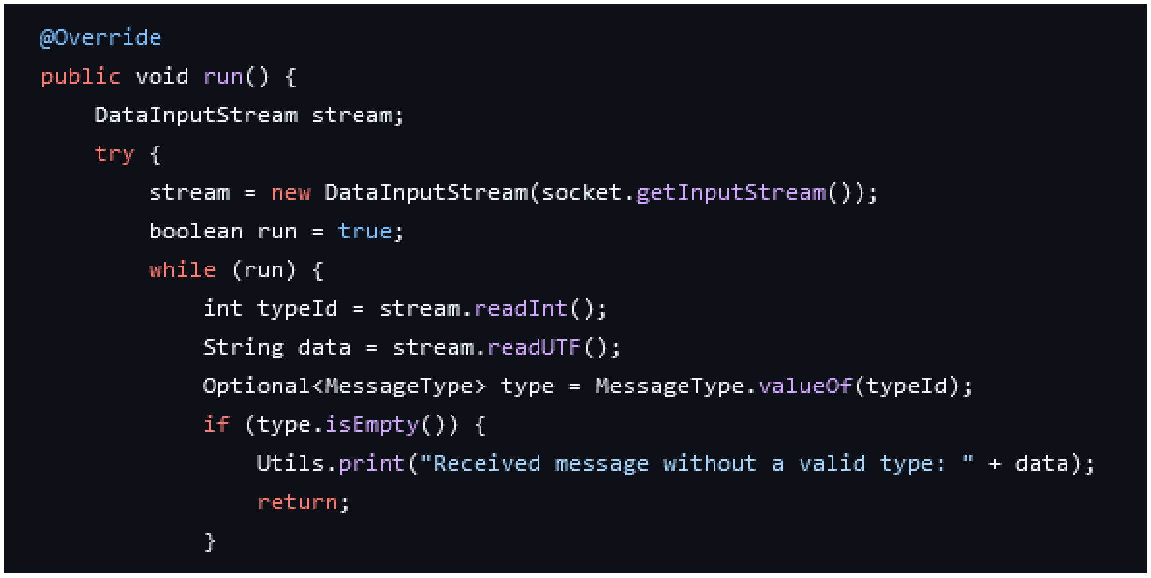
3.2 Server Setup



```java
public static void main(String[] args) throws IOException {
    Utils.print("Starting server...");
    System.setProperty( "javax.net.ssl.keyStore", "C:\\cs2sns-thechase\\thechase.jks" );
    System.setProperty( "javax.net.ssl.trustStore", "C:\\cs2sns-thechase\\thechase.jks" );
    System.setProperty( "javax.net.ssl.keyStorePassword", "hotten" );

    SSLServerSocketFactory sslServerSocketFactory = (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
    SSLServerSocket ss = (SSLServerSocket) sslServerSocketFactory.createServerSocket(17777);
    ss.setEnabledProtocols( new String[]{"TLSv1.3", "TLSv1.2"} );

    Utils.print("Awaiting connections.");
    while(chaser == null || player == null) {
        if (chaser == null) {
            chaser = ss.accept();
            chaserStream = new DataOutputStream(chaser.getOutputStream());
            Utils.print("Chaser is connected.");
            sendToChaser(MessageType.ROLE_DESIGNATION, "chaser");
            sendToChaser(MessageType.TEXT, "Welcome. Please wait. You are the chaser.");
            new ServerDataReceiveThread(chaser, true).start();
        } else {
            player = ss.accept();
            playerStream = new DataOutputStream(player.getOutputStream());
            Utils.print("Player is connected.");
            sendToPlayer(MessageType.ROLE_DESIGNATION, "player");
            sendToPlayer(MessageType.TEXT, "Welcome. Please wait. You are the player.");
            new ServerDataReceiveThread(player, false).start();
        }
    }

    start();
}
```

Figure 3: Server.java

To start, an SSL socket server is created using the systems key/trust store and a specified password. The application uses TLS v1.2 and v1.3. When a client is securely connected via TLS, the server sends a ”ROLE DESIGNATION” message to the client informing them of their role, alongside a welcome message. The server then creates a ”server data receive thread” which listens for messages from the client(s) and performs actions based on the data.

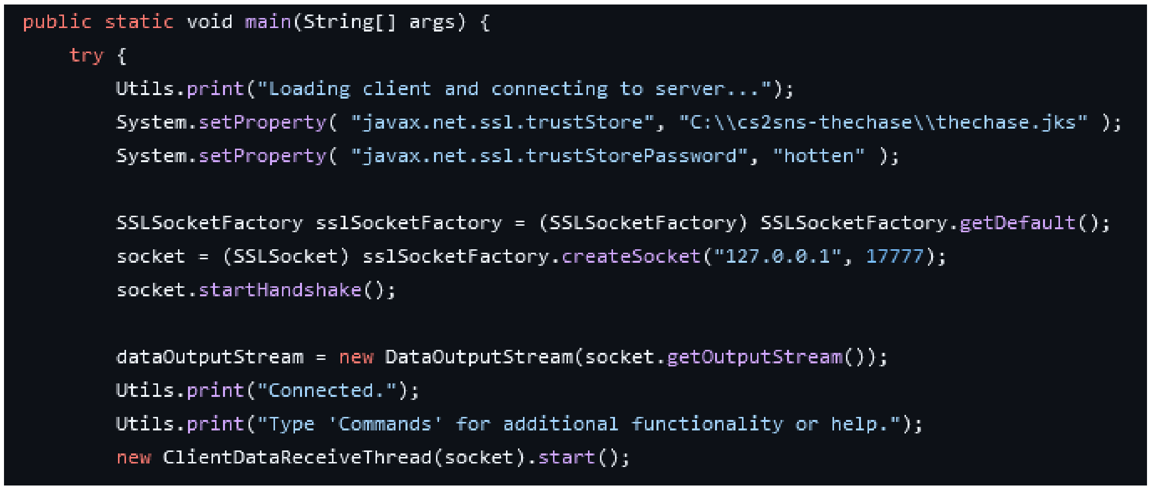


```java
@Override
public void run() {
    DataInputStream stream;
    try {
        stream = new DataInputStream(socket.getInputStream());
        boolean run = true;
        while (run) {
            int typeId = stream.readInt();
            String data = stream.readUTF();
            Optional<MessageType> type = MessageType.valueOf(typeId);
            if (type.isEmpty()) {
                Utils.print("Received message without a valid type: " + data);
                return;
            }
```

Figure 4: ServerDataReceiveThread.java

While the client is connected, it awaits the message integer and the associated data to be received. It then checks to ensure that the integer received was a valid type. Based on the integer received, it will perform the appropriate action.

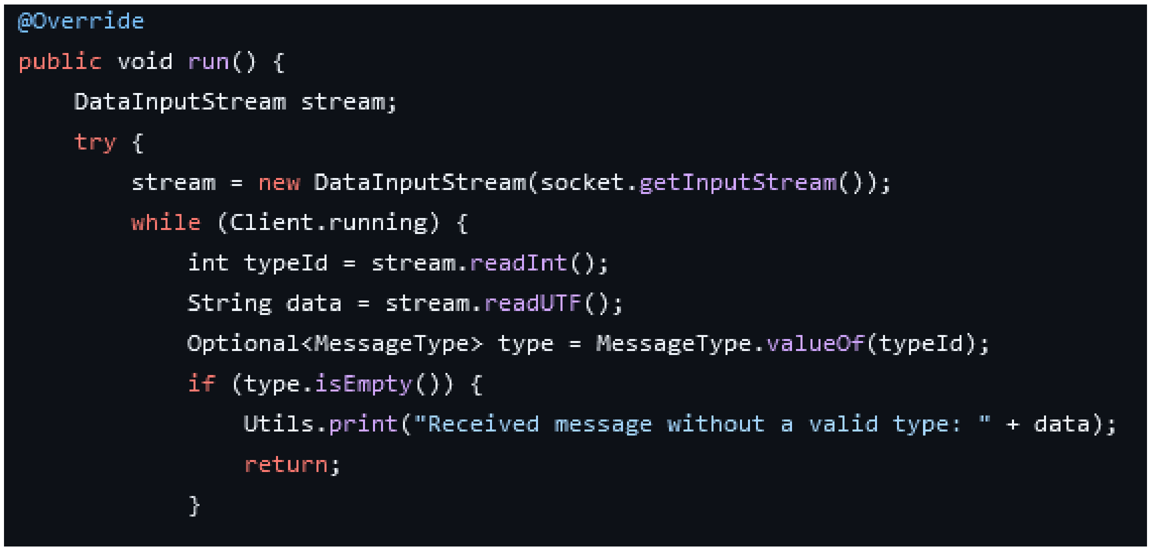
3.3 Client Setup



```java
public static void main(String[] args) {
    try {
        Utils.print("Loading client and connecting to server...");
        System.setProperty( "javax.net.ssl.trustStore", "C:\\cs2sns-thechase\\thechase.jks" );
        System.setProperty( "javax.net.ssl.trustStorePassword", "hotten" );

        SSLSocketFactory sslSocketFactory = (SSLSocketFactory) SSLSocketFactory.getDefault();
        socket = (SSLSocket) sslSocketFactory.createSocket("127.0.0.1", 17777);
        socket.startHandshake();

        dataOutputStream = new DataOutputStream(socket.getOutputStream());
        Utils.print("Connected.");
        Utils.print("Type 'Commands' for additional functionality or help.");
        new ClientDataReceiveThread(socket).start();
```

Figure 5: Client.java

For the client to connect, it will load the same key/trust store data with the specified password to create a secure SSL socket connection to the server. Once connected, the client will start a ”client data receive thread” which listens for messages from the server and performs action based on the data.

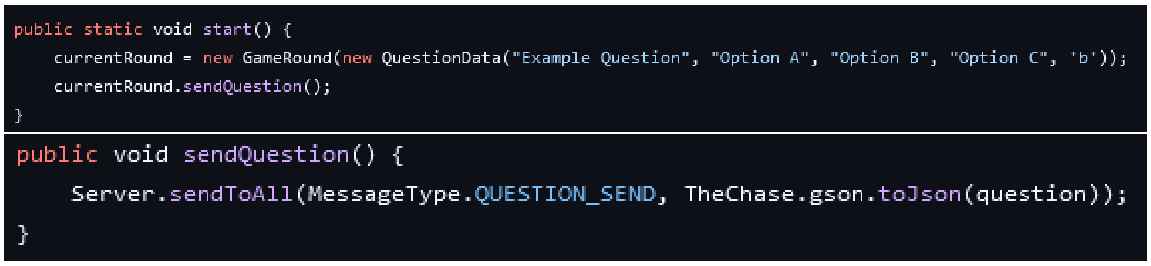


```java
@Override
public void run() {
    DataInputStream stream;
    try {
        stream = new DataInputStream(socket.getInputStream());
        while (Client.running) {
            int typeId = stream.readInt();
            String data = stream.readUTF();
            Optional<MessageType> type = MessageType.valueOf(typeId);
            if (type.isEmpty()) {
                Utils.print("Received message without a valid type: " + data);
                return;
            }
```

Figure 6: ClientDataReceiveThread.java

Similar to the server's data receive thread, it awaits messages from the servers and validates the received integers. Based on the integer received, it will perform the appropriate action.

3.4 Game Logic

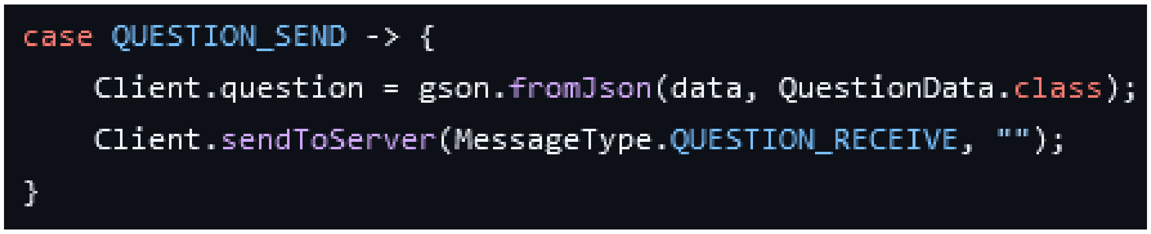


```java
public static void start() {
    currentRound = new GameRound(new QuestionData("Example Question", "Option A", "Option B", "Option C", 'b'));
    currentRound.sendQuestion();
}
```

```java
public void sendQuestion() {
    Server.sendToAll(MessageType.QUESTION_SEND, TheChase.gson.toJson(question));
}
```

Figure 7: Server.java and GameRound.java

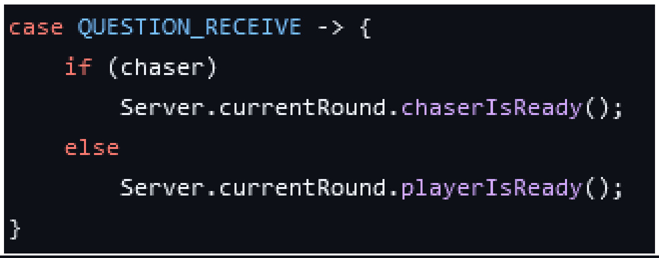
Now the clients are connected, the server creates a new "game round" and sends the next question via the "QUESTION SEND" message.



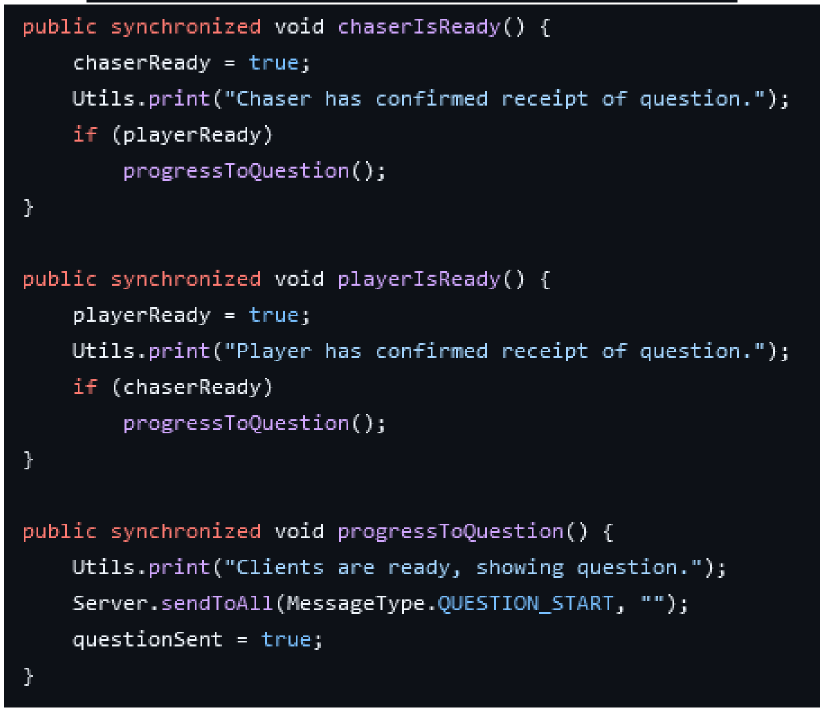
```java
case QUESTION_SEND -> {
    Client.question = gson.fromJson(data, QuestionData.class);
    Client.sendToServer(MessageType.QUESTION_RECEIVE, "");
}
```

Figure 8: ClientDataReceiveThread.java

The client has now received the question. For fairness and ensuring that both clients have received the question, the client sends back to the server a ”QUESTION RECEIVE” and does not yet display the question.



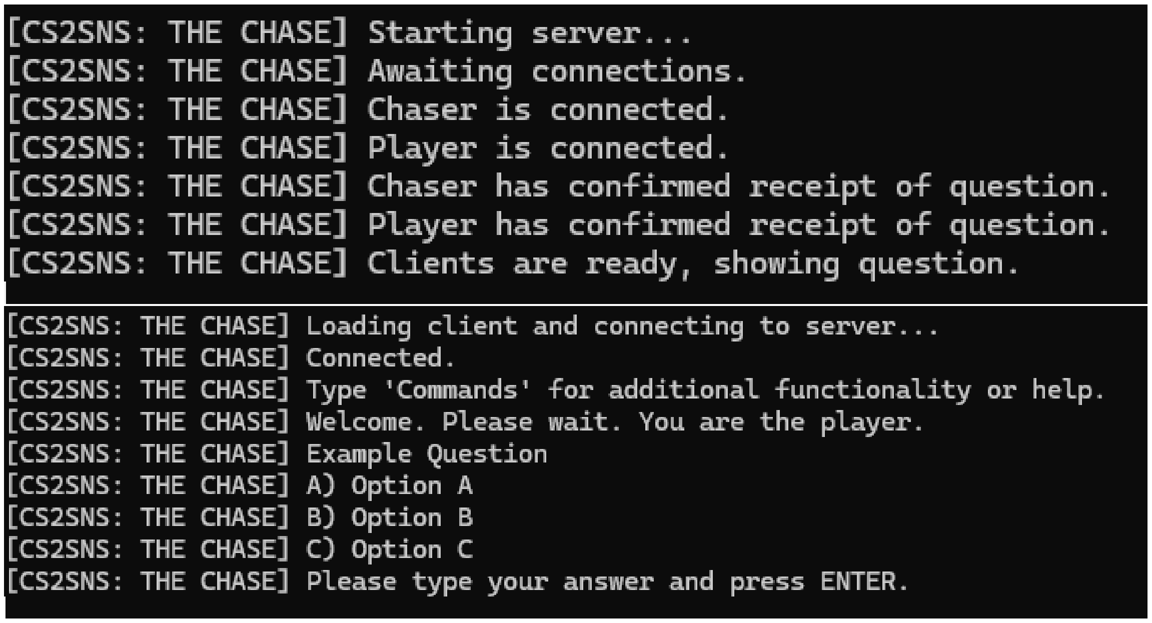
```
case QUESTION_RECEIVE -> {
    if (chaser)
        Server.currentRound.chaserIsReady();
    else
        Server.currentRound.playerIsReady();
}
```



```
public synchronized void chaserIsReady() {
    chaserReady = true;
    Utils.print("Chaser has confirmed receipt of question.");
    if (playerReady)
        progressToQuestion();
}

public synchronized void playerIsReady() {
    playerReady = true;
    Utils.print("Player has confirmed receipt of question.");
    if (chaserReady)
        progressToQuestion();
}

public synchronized void progressToQuestion() {
    Utils.print("Clients are ready, showing question.");
    Server.sendToAll(MessageType.QUESTION_START, "");
    questionSent = true;
}
```

Figure 9: ServerDataReceiveThread.java and GameRound.java

When both clients have confirmed receipt of the question, the server sends a ''QUESTION START'' message which prompts the clients to display the message.

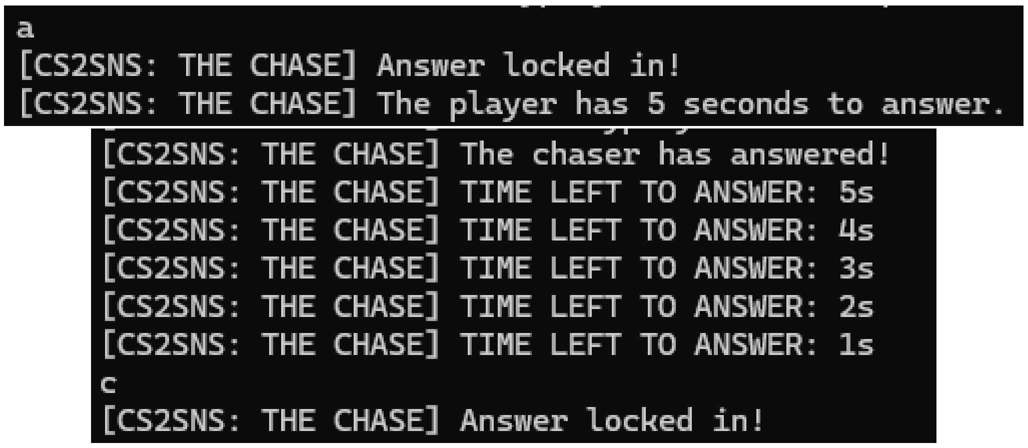


```
[CS2SNS: THE CHASE] Starting server...
[CS2SNS: THE CHASE] Awaiting connections.
[CS2SNS: THE CHASE] Chaser is connected.
[CS2SNS: THE CHASE] Player is connected.
[CS2SNS: THE CHASE] Chaser has confirmed receipt of question.
[CS2SNS: THE CHASE] Player has confirmed receipt of question.
[CS2SNS: THE CHASE] Clients are ready, showing question.
```

```
[CS2SNS: THE CHASE] Loading client and connecting to server...
[CS2SNS: THE CHASE] Connected.
[CS2SNS: THE CHASE] Type 'Commands' for additional functionality or help.
[CS2SNS: THE CHASE] Welcome. Please wait. You are the player.
[CS2SNS: THE CHASE] Example Question
[CS2SNS: THE CHASE] A) Option A
[CS2SNS: THE CHASE] B) Option B
[CS2SNS: THE CHASE] C) Option C
[CS2SNS: THE CHASE] Please type your answer and press ENTER.
```

Figure 10: Server and client's view when a question is sent

This is the view of the server, which has logs on when players connect, when they confirm receipt of questions and when they're sent. From the client's side, they are informed of their role and are displayed the question when instructed by the server.

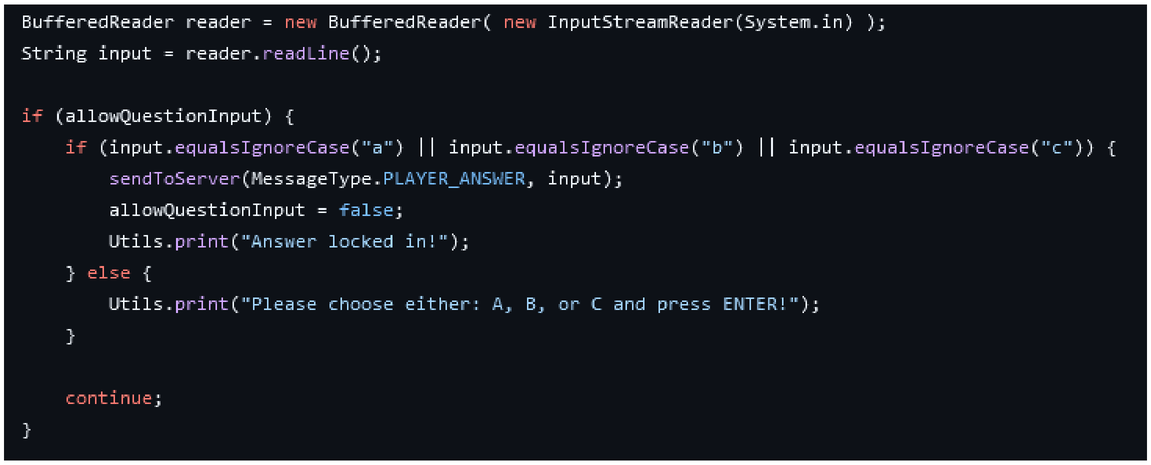


```
a
[CS2SNS: THE CHASE] Answer locked in!
[CS2SNS: THE CHASE] The player has 5 seconds to answer.
```

```
[CS2SNS: THE CHASE] The chaser has answered!
[CS2SNS: THE CHASE] TIME LEFT TO ANSWER: 5s
[CS2SNS: THE CHASE] TIME LEFT TO ANSWER: 4s
[CS2SNS: THE CHASE] TIME LEFT TO ANSWER: 3s
[CS2SNS: THE CHASE] TIME LEFT TO ANSWER: 2s
[CS2SNS: THE CHASE] TIME LEFT TO ANSWER: 1s
c
[CS2SNS: THE CHASE] Answer locked in!
```

Figure 11: Chaser and player's view when the timer starts

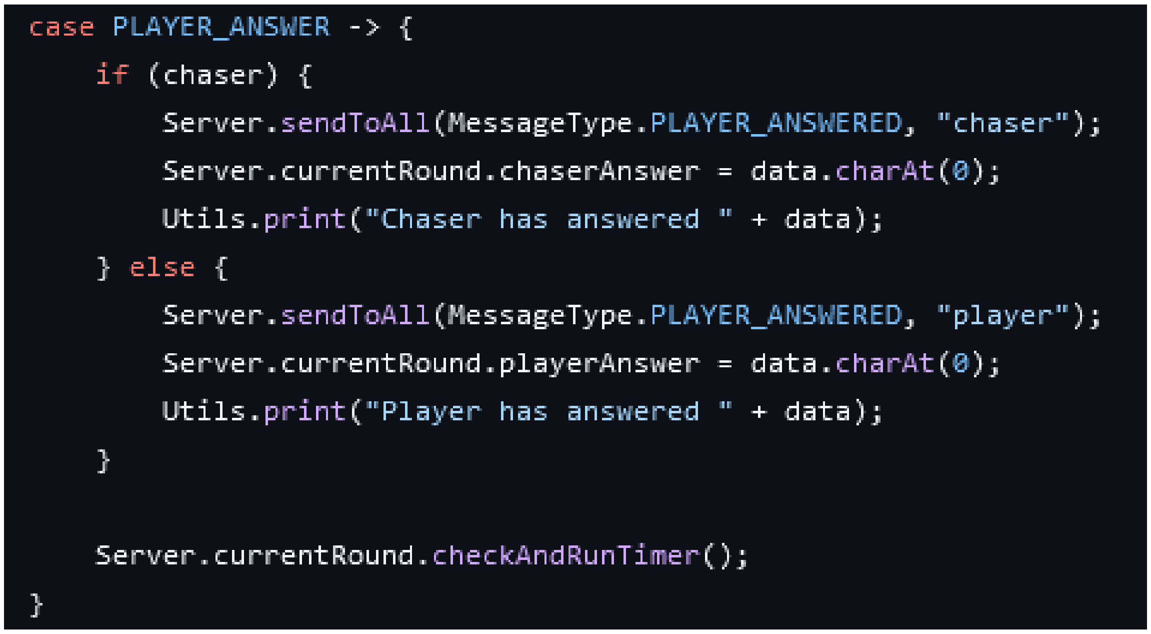
When one player answers, the other player has 5 seconds to answer. If they don't answer in time, they are locked out and will not score points this round.



```java
BufferedReader reader = new BufferedReader( new InputStreamReader(System.in) );
String input = reader.readLine();

if (allowQuestionInput) {
    if (input.equalsIgnoreCase("a") || input.equalsIgnoreCase("b") || input.equalsIgnoreCase("c")) {
        sendToServer(MessageType.PLAYER_ANSWER, input);
        allowQuestionInput = false;
        Utils.print("Answer locked in!");
    } else {
        Utils.print("Please choose either: A, B, or C and press ENTER!");
    }

    continue;
}
```

Figure 12: Client.java

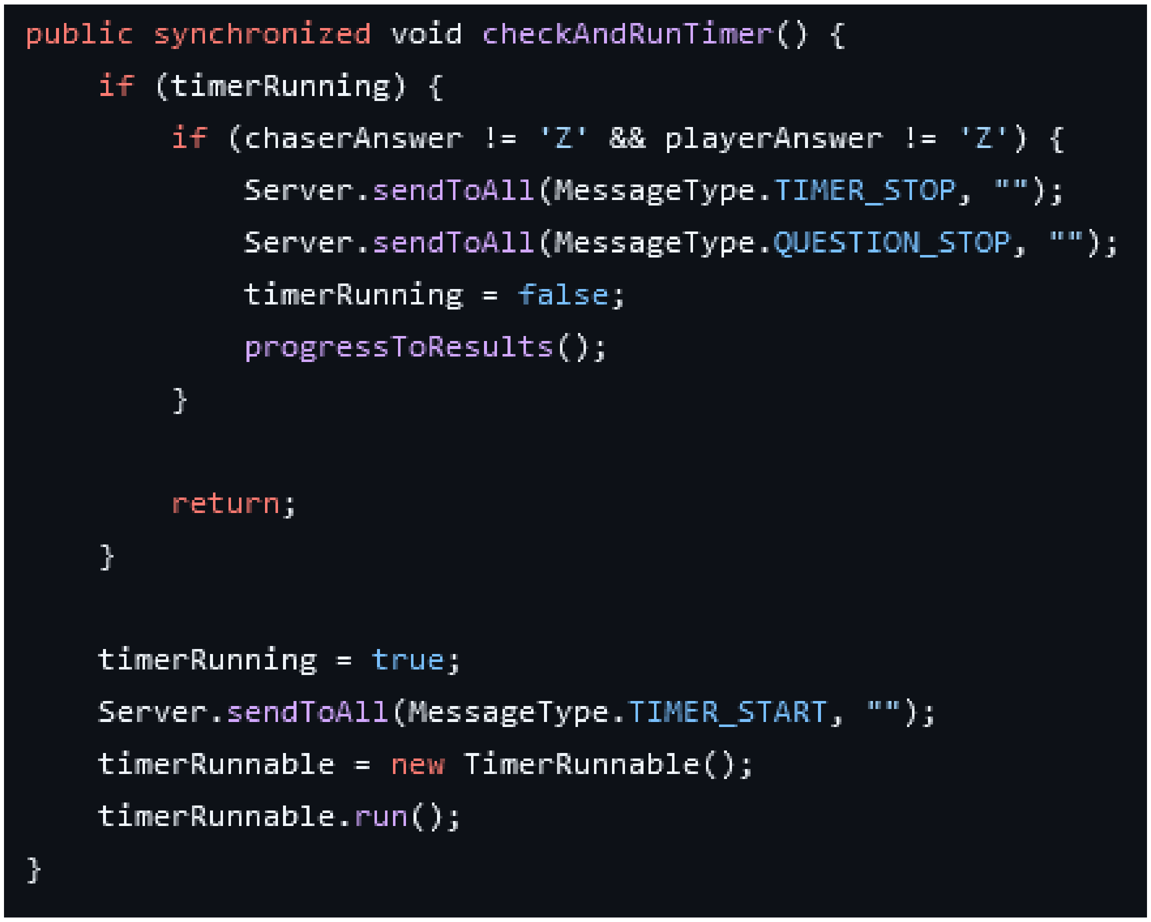
This is the logic for a player entering an answer. Their input is checked to ensure it is either A, B or C. If their input is valid, it is sent to the server with a "PLAYER ANSWER" message.



```java
case PLAYER_ANSWER -> {
    if (chaser) {
        Server.sendToAll(MessageType.PLAYER_ANSWERED, "chaser");
        Server.currentRound.chaserAnswer = data.charAt(0);
        Utils.print("Chaser has answered " + data);
    } else {
        Server.sendToAll(MessageType.PLAYER_ANSWERED, "player");
        Server.currentRound.playerAnswer = data.charAt(0);
        Utils.print("Player has answered " + data);
    }

    Server.currentRound.checkAndRunTimer();
}
```

Figure 13: ServerDataReceiveThread.java

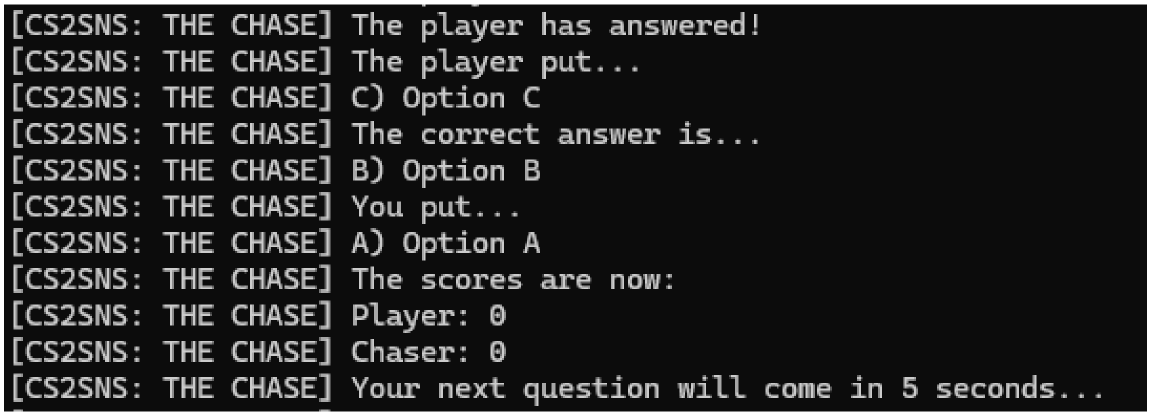
When the server receives the answer data, it is stored and then the server checks to see what must happen next.



```java
public synchronized void checkAndRunTimer() {
    if (timerRunning) {
        if (chaserAnswer != 'Z' && playerAnswer != 'Z') {
            Server.sendToAll(MessageType.TIMER_STOP, "");
            Server.sendToAll(MessageType.QUESTION_STOP, "");
            timerRunning = false;
            progressToResults();
        }

        return;
    }

    timerRunning = true;
    Server.sendToAll(MessageType.TIMER_START, "");
    timerRunnable = new TimerRunnable();
    timerRunnable.run();
}
```

Figure 14: GameRound.java

If only one player has answered, the server will send the "TIMER START" message to start the timer music for the clients and start a server-side five-second countdown. If both players have answered whilst the timer is running, it is stopped and clients are sent the "TIMER STOP" and "QUESTION STOP" to stop the timer music and the question input respectively.



```
[CS2SNS: THE CHASE] The player has answered!
[CS2SNS: THE CHASE] The player put...
[CS2SNS: THE CHASE] C) Option C
[CS2SNS: THE CHASE] The correct answer is...
[CS2SNS: THE CHASE] B) Option B
[CS2SNS: THE CHASE] You put...
[CS2SNS: THE CHASE] A) Option A
[CS2SNS: THE CHASE] The scores are now:
[CS2SNS: THE CHASE] Player: 0
[CS2SNS: THE CHASE] Chaser: 0
[CS2SNS: THE CHASE] Your next question will come in 5 seconds...
```

Figure 15: Chaser's view after a question

After all players have answered or timed out, the results screen is show at two second intervals announcing what each player put and what the correct answer is. If a player gets the answer correct, they get a point added to their score. Whoever reaches a score of three first is crowned the winner.

4 Application Security

To ensure the best player experience, and to eliminate the risk of hackers cheating at the game, we have implemented a secure socket layer (SSL) and transport layer security (TLS).

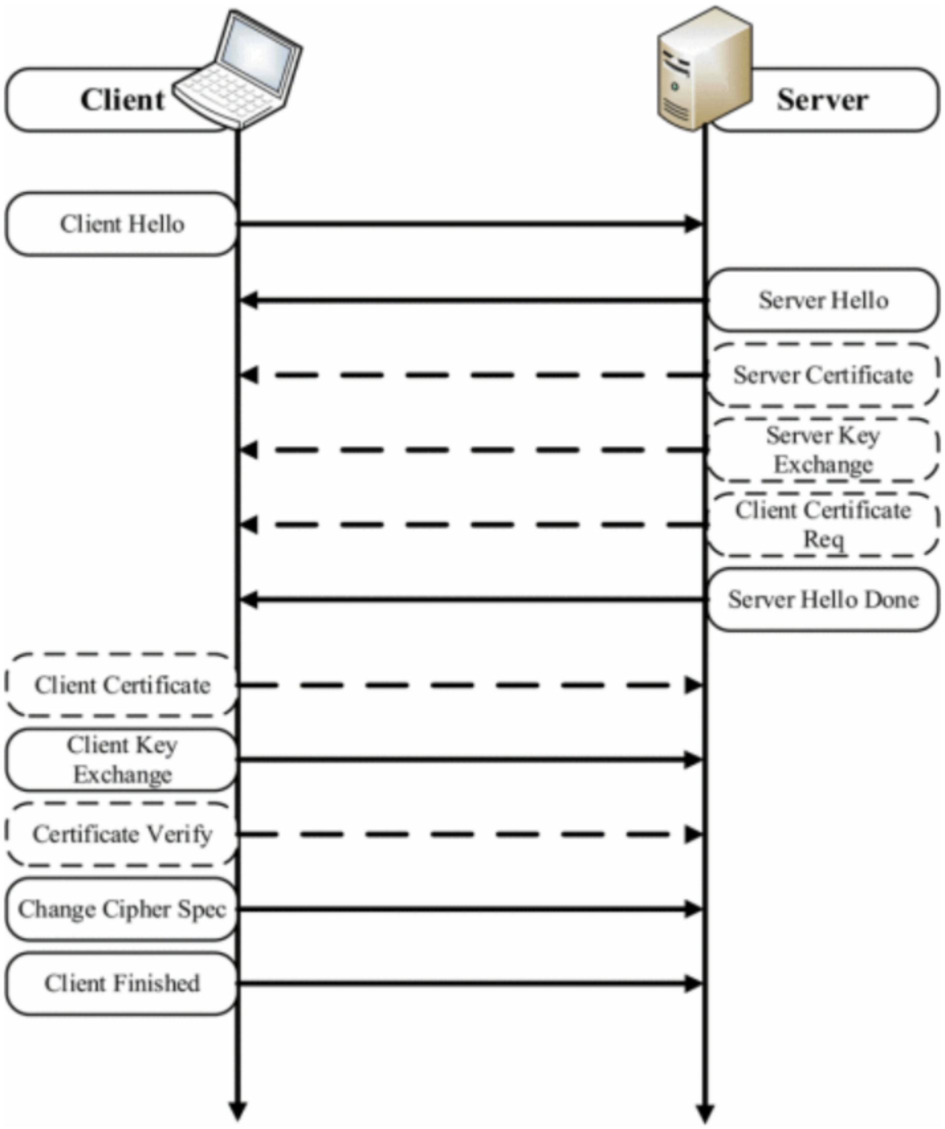


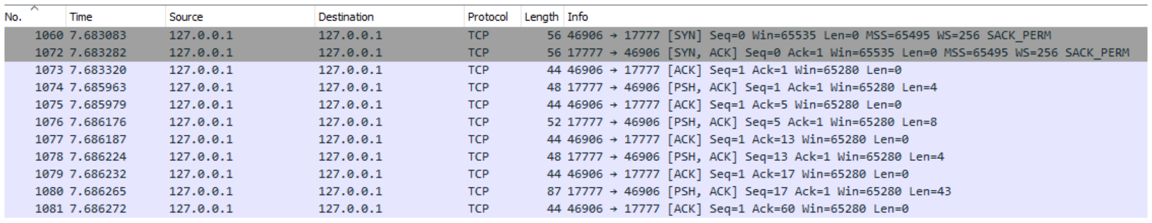
Figure 16: SSL/TLS full/abbreviated handshake protocol (Kim et al., 2015)

This process shows an SSL handshake process which shows how the client and server communicate and establish a secure, encrypted connection. We then wanted to test if our packets are encrypted. We decided to use Wireshark to use this. Sandhya et al. (2017) explains the importance of Wireshark as it is an ethical hacking tool that can "reveal the flaws in the system security at the user authentication level." which makes it perfect for testing our SSL encryption and gain confirmation of the functionality.

5 WireGuard Analysis

First, we will review the packets to/from our application without any form of encryption enabled.

5.1 Analysis without Encryption



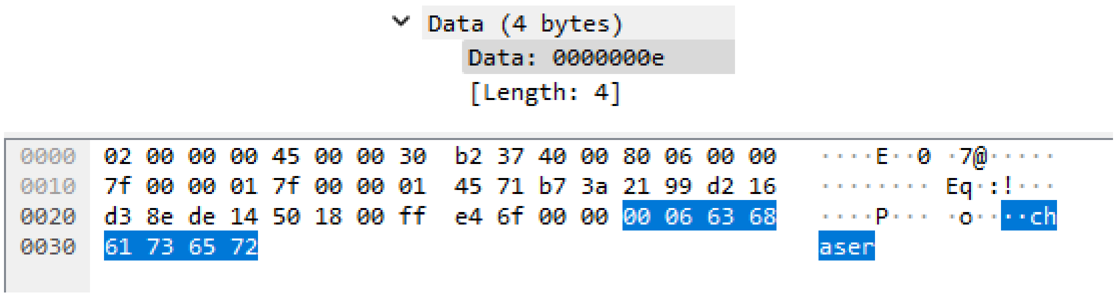
No.	Time	Source	Destination	Protocol	Length	Info
1060	7.683083	127.0.0.1	127.0.0.1	TCP	56	46906 → 17777 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
1072	7.683282	127.0.0.1	127.0.0.1	TCP	56	17777 → 46906 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
1073	7.683320	127.0.0.1	127.0.0.1	TCP	44	46906 → 17777 [ACK] Seq=1 Ack=1 Win=65280 Len=0
1074	7.685963	127.0.0.1	127.0.0.1	TCP	48	17777 → 46906 [PSH, ACK] Seq=1 Ack=1 Win=65280 Len=4
1075	7.685979	127.0.0.1	127.0.0.1	TCP	44	46906 → 17777 [ACK] Seq=1 Ack=5 Win=65280 Len=0
1076	7.686176	127.0.0.1	127.0.0.1	TCP	52	17777 → 46906 [PSH, ACK] Seq=5 Ack=1 Win=65280 Len=8
1077	7.686187	127.0.0.1	127.0.0.1	TCP	44	46906 → 17777 [ACK] Seq=1 Ack=13 Win=65280 Len=0
1078	7.686224	127.0.0.1	127.0.0.1	TCP	48	17777 → 46906 [PSH, ACK] Seq=13 Ack=1 Win=65280 Len=4
1079	7.686232	127.0.0.1	127.0.0.1	TCP	44	46906 → 17777 [ACK] Seq=1 Ack=17 Win=65280 Len=0
1080	7.686265	127.0.0.1	127.0.0.1	TCP	87	17777 → 46906 [PSH, ACK] Seq=17 Ack=1 Win=65280 Len=43
1081	7.686272	127.0.0.1	127.0.0.1	TCP	44	46906 → 17777 [ACK] Seq=1 Ack=60 Win=65280 Len=0

Figure 17: Wireshark view of a client connecting to the server

Here we can see a regular TCP handshake following the SYN, SYN-ACK, and ACK process on the first three packets. This process synchronizes the two endpoints and ensures they're both ready to communicate (GeeksforGeeks, 2024).

5.1.1 Chase Communication System

Looking at the next few packets, we can identify a "ROLE DESIGNATION" message being sent from the server to the client.

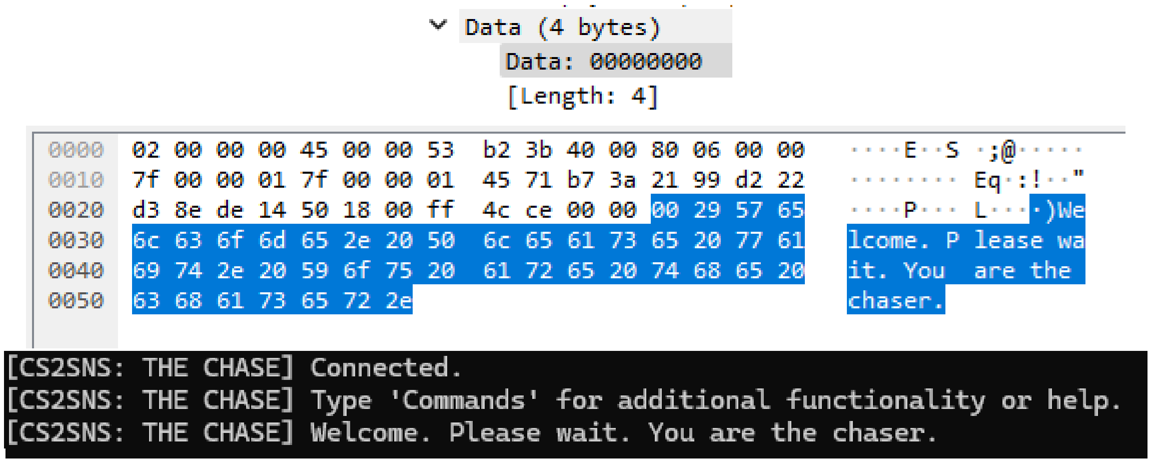


```
v Data (4 bytes)
    Data: 0000000e
    [Length: 4]

0000  02 00 00 00 45 00 00 30  b2 37 40 00 80 06 00 00   ····E··0 ·7@·····
0010  7f 00 00 01 7f 00 00 01  45 71 b7 3a 21 99 d2 16   ········ Eq·:!···
0020  d3 8e de 14 50 18 00 ff  e4 6f 00 00 00 06 63 68   ····P··· ·o····ch
0030  61 73 65 72                                        aser
```

Figure 18: Role Designation packets

On the first packet, we can see the hex number 0e being sent. This corresponds to fourteen in decimal, which is the same number used for the role designation message. After this packet, we see the text "chaser" being sent. This aligns with the data sent with the role designation message, as the server will send either "chaser" or "player" to inform the client of their role.

After the role is sent, the server sends a "TEXT" message welcoming the client to the game. This can also be observed with the initial value of zero being sent before the text. Zero is the integer value associated with the text message.

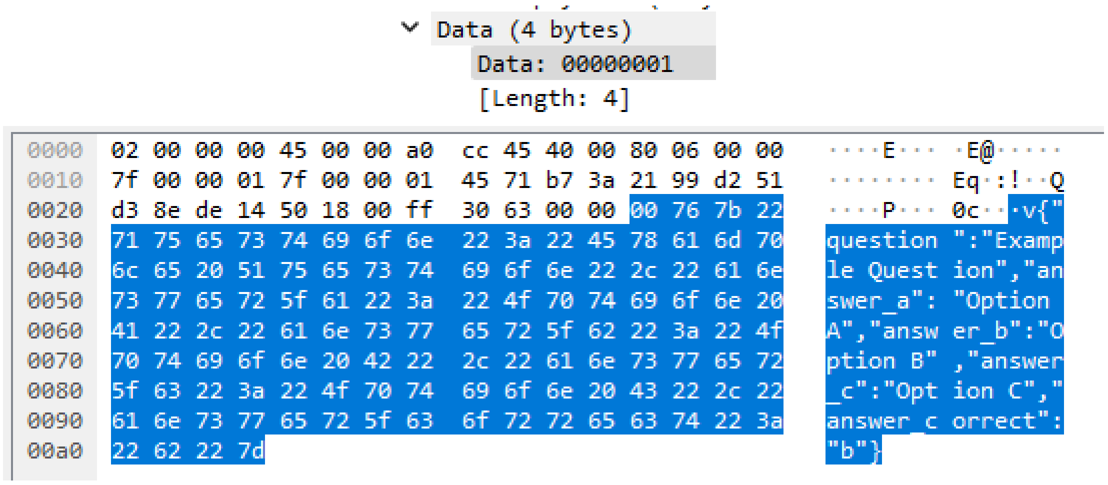


```
Data (4 bytes)
    Data: 00000000
    [Length: 4]

0000  02 00 00 00 45 00 00 53  b2 3b 40 00 80 06 00 00   ····E··S ·;@·····
0010  7f 00 00 01 7f 00 00 01  45 71 b7 3a 21 99 d2 22   ········ Eq·:!··"
0020  d3 8e de 14 50 18 00 ff  4c ce 00 00 00 29 57 65   ····P··· L····)We
0030  6c 63 6f 6d 65 2e 20 50  6c 65 61 73 65 20 77 61   lcome. P lease wa
0040  69 74 2e 20 59 6f 75 20  61 72 65 20 74 68 65 20   it. You  are the
0050  63 68 61 73 65 72 2e                               chaser.
```

```
[CS2SNS: THE CHASE] Connected.
[CS2SNS: THE CHASE] Type 'Commands' for additional functionality or help.
[CS2SNS: THE CHASE] Welcome. Please wait. You are the chaser.
```

Figure 19: Text message packets

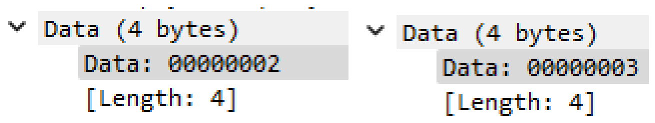
When a second player has connected, the server sends out a "QUESTION SEND" message containing the question to display. This is represented with an integer of one, followed by the question in a JSON format that the client can encode.



```
Data (4 bytes)
    Data: 00000001
    [Length: 4]

0000  02 00 00 00 45 00 00 a0  cc 45 40 00 80 06 00 00   ····E··· ·E@·····
0010  7f 00 00 01 7f 00 00 01  45 71 b7 3a 21 99 d2 51   ········ Eq·:!··Q
0020  d3 8e de 14 50 18 00 ff  30 63 00 00 00 76 7b 22   ····P··· 0c···v{"
0030  71 75 65 73 74 69 6f 6e  22 3a 22 45 78 61 6d 70   question ":"Examp
0040  6c 65 20 51 75 65 73 74  69 6f 6e 22 2c 22 61 6e   le Quest ion","an
0050  73 77 65 72 5f 61 22 3a  22 4f 70 74 69 6f 6e 20   swer_a": "Option
0060  41 22 2c 22 61 6e 73 77  65 72 5f 62 22 3a 22 4f   A","answ er_b":"O
0070  70 74 69 6f 6e 20 42 22  2c 22 61 6e 73 77 65 72   ption B" ,"answer
0080  5f 63 22 3a 22 4f 70 74  69 6f 6e 20 43 22 2c 22   _c":"Opt ion C","
0090  61 6e 73 77 65 72 5f 63  6f 72 72 65 63 74 22 3a   answer_c orrect":
00a0  22 62 22 7d                                        "b"}
```

Figure 20: Question Send packets

The client then sends a "QUESTION RECEIVE" packet, represented by a two. When both clients have sent this message, the server sends a "QUESTION START", represented by a three. Once this message is received, the clients display the question ready for answering.



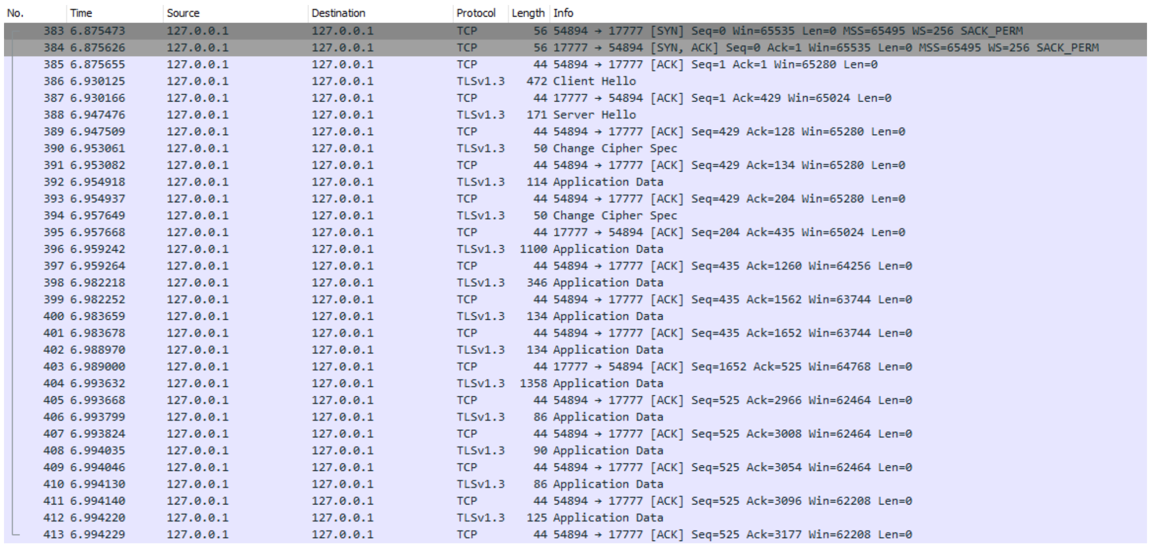
```
Data (4 bytes)
    Data: 00000002
    [Length: 4]
```

```
Data (4 bytes)
    Data: 00000003
    [Length: 4]
```

Figure 21: Question Receive and Start packets

5.2 Analysis with Encryption

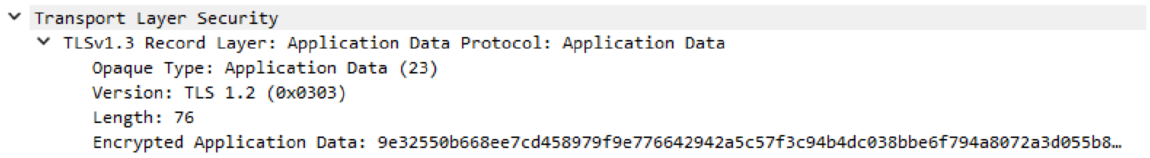
With SSL and TLS enabled, our WireShark looks very different.



No.	Time	Source	Destination	Protocol	Length	Info
383	6.875473	127.0.0.1	127.0.0.1	TCP	56	54894 → 17777 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
384	6.875626	127.0.0.1	127.0.0.1	TCP	56	17777 → 54894 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM
385	6.875655	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=1 Ack=1 Win=65280 Len=0
386	6.930125	127.0.0.1	127.0.0.1	TLSv1.3	472	Client Hello
387	6.930166	127.0.0.1	127.0.0.1	TCP	44	17777 → 54894 [ACK] Seq=1 Ack=429 Win=65024 Len=0
388	6.947476	127.0.0.1	127.0.0.1	TLSv1.3	171	Server Hello
389	6.947509	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=429 Ack=128 Win=65280 Len=0
390	6.953061	127.0.0.1	127.0.0.1	TLSv1.3	50	Change Cipher Spec
391	6.953082	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=429 Ack=134 Win=65280 Len=0
392	6.954918	127.0.0.1	127.0.0.1	TLSv1.3	114	Application Data
393	6.954937	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=429 Ack=204 Win=65280 Len=0
394	6.957649	127.0.0.1	127.0.0.1	TLSv1.3	50	Change Cipher Spec
395	6.957668	127.0.0.1	127.0.0.1	TCP	44	17777 → 54894 [ACK] Seq=204 Ack=435 Win=65024 Len=0
396	6.959242	127.0.0.1	127.0.0.1	TLSv1.3	1100	Application Data
397	6.959264	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=435 Ack=1260 Win=64256 Len=0
398	6.982218	127.0.0.1	127.0.0.1	TLSv1.3	346	Application Data
399	6.982252	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=435 Ack=1562 Win=63744 Len=0
400	6.983659	127.0.0.1	127.0.0.1	TLSv1.3	134	Application Data
401	6.983678	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=435 Ack=1652 Win=63744 Len=0
402	6.988970	127.0.0.1	127.0.0.1	TLSv1.3	134	Application Data
403	6.989000	127.0.0.1	127.0.0.1	TCP	44	17777 → 54894 [ACK] Seq=1652 Ack=525 Win=64768 Len=0
404	6.993632	127.0.0.1	127.0.0.1	TLSv1.3	1358	Application Data
405	6.993668	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=525 Ack=2966 Win=62464 Len=0
406	6.993799	127.0.0.1	127.0.0.1	TLSv1.3	86	Application Data
407	6.993824	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=525 Ack=3008 Win=62464 Len=0
408	6.994035	127.0.0.1	127.0.0.1	TLSv1.3	90	Application Data
409	6.994046	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=525 Ack=3054 Win=62464 Len=0
410	6.994130	127.0.0.1	127.0.0.1	TLSv1.3	86	Application Data
411	6.994140	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=525 Ack=3096 Win=62208 Len=0
412	6.994220	127.0.0.1	127.0.0.1	TLSv1.3	125	Application Data
413	6.994229	127.0.0.1	127.0.0.1	TCP	44	54894 → 17777 [ACK] Seq=525 Ack=3177 Win=62208 Len=0

Figure 22: Wireshark view of a client connecting to the server with TLS

We can still see our TCP handshake, but now we can also see our TLS handshake. If we look at any of the packets sent, they all contain "Encrypted Application Data", making it impossible to read the data being transmitted.



```
Transport Layer Security
  TLSv1.3 Record Layer: Application Data Protocol: Application Data
    Opaque Type: Application Data (23)
    Version: TLS 1.2 (0x0303)
    Length: 76
    Encrypted Application Data: 9e32550b668ee7cd458979f9e776642942a5c57f3c94b4dc038bbe6f794a8072a3d055b8…
```

Figure 23: The data within a packet sent from the server

This adds vital security to our application, ensuring that hackers are unable to intercept data being sent to/from the server making it impossible to cheat or exploit.

6 Other Potential Security Features

In the future, we would implement additional security measures for robust protection of client-server communications. One of these features would be a passkey. Each client would be provided with a unique, random string of text issued by the server upon connection, which must be present and allows the server to verify that the communication is coming from the verified client. It would also be encrypted using Transport Layer Security (TLS), which adds protection against eavesdropping such as sniffing, impersonation, and tampering.

In addition, we would also add dynamic firewalls, as they have an advantage over traditional ones since they can modify rules in real time depending on authenticated connections. When the client has successfully connected, the firewall validates the client's IP address and only allows for traffic to go in and from there. It ensures that only authorised users can exchange data with the server, adding an additional layer to blocking unauthorised access.

References

GeeksforGeeks (2024), 'Tcp 3-way handshake process'. Accessed: 22 December 2024. **URL:** *https://www.geeksforgeeks.org/tcp-3-way-handshake-process/*

Kim, S.-M., Goo, Y.-H., Kim, M.-S., Choi, S.-G. and Choi, M.-J. (2015), A method for service identification of ssl/tls encrypted traffic with the relation of session id and server ip, *in* '2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)', pp. 487–490. Available at: https://ieeexplore.ieee.org/document/7275373.

Sandhya, S., Purkayastha, S., Joshua, E. and Deep, A. (2017), Assessment of website security by penetration testing using wireshark, *in* '2017 4th International Conference on Advanced Computing and Communication Systems (ICACCS)', pp. 1–4. Available at: https://ieeexplore.ieee.org/document/8014711.