

Contents

Paperclip	3
Abstract	3
1 Introduction	3
1.1 Problem Description	3
1.2 Objectives	3
1.2.1 Primary Objectives	3
1.2.2 Secondary Objectives	3
1.3 Beneficiaries	3
1.4 Assumptions and limitations	4
2 Output Summary	4
3 Literature Review	4
3.1 Network Protocols	4
3.1.1 TCP	4
3.1.2 UDP	5
3.1.3 Comparison	6
3.2 RESTful API	6
3.3 Security Algorithms	7
3.3.1 TLS	7
3.3.2 Session Keys	7
3.3.3 Authentication	8
4 Method	8
4.0 Tools	8
4.0.1 Programming Language	8
4.0.2 Database	8
4.0.3 IDE	8
4.0.4 Source Control	8
4.1 Methodology	9
4.2 Analysis	9
4.3 Design	9
4.3.1 Packet Specification	9
4.3.2 Database Models	10
4.3.3 API Specification	10
4.4 Implementation	10
4.4.1 Iteration 1	10
4.4.2 Iteration 2	11
4.4.3 Iteration 3	11
4.4.4 Iteration 4	12
4.5 Tests	13
4.6 Reused Code and Tutorials	13
5 Results	13
5.3 Design	13
5.3.1 Packet Specification	13
5.3.2 Database Models	13

5.3.3 API Specification	14
5.4 Implementation	14
5.4.1 Iteration 1	14
5.4.2 Iteration 2	19
5.4.3 Iteration 3	32
5.4.4 Iteration 4	41
5.5 Tests	70
6 Conclusion	71
6.1 Project Objective	71
6.1.1 Objective One	71
6.1.2 Objective Two	71
6.1.3 Secondary Objectives	71
6.2 Feature Evaluation	72
6.2.1 Reflection on Handshake Finished	72
6.2.2 Port Selection and Assignment	72
6.2.3 Packet ERRORS	72
6.2.4 Automatic Game Importing	72
6.2.5 Congestion Control	72
6.2.6 Features Cut for Time	72
7 Glossary	73
8 References	75
9 Appendices	77
9.1 Project Definition Document	77
Cover Sheet	77
Proposal	77
9.2 Deployment Guide	81
Requirements	81
mysql	82
Python	82
9.3 Package	83
9.3.1 udp	83
9.3.2 server	151
9.3.3 rps	164
9.3.4 client	175
9.3.8 inputimout	193
9.4 Packet Specification	196
Packets	196
Behavior	201
9.5 API Specification	204
9.6 ERD Diagram	205

Paperclip

Abstract

This project attempts to define and implement networking tools including a lightweight protocol for use with game technology and an account and game session management server.

1 Introduction

1.1 Problem Description

When creating a multiplayer game, every developer must integrate a few key features. Even though every multilayer game shares these same key features, a developer must build these from scratch. This is time consuming and means that a developer is limited in the amount of initial development they can put into their game. Though some implementations of a custom networking protocol exist (described in the Literature Review [3]), no implementation provides both a variety of rich-features with full flexibility in when/which features are used.

1.2 Objectives

1.2.1 Primary Objectives

1. Create a scalable system for managing user accounts and inter-account interactions including matchmaking and friends.
2. Create a custom UDP protocol that implements key features required for game communication missing from vanilla UDP. This includes features to improve reliability and security.

1.2.2 Secondary Objectives The secondary objectives are split into sub objectives as follows:

Secondary Objectives
Turn-based game demo
Real-time game demo
Large packet size limit
Authenticated Communication
Secure Communication
Reliable Communication
Lightweight Communication
Modularity

1.3 Beneficiaries

The project is intended to be used by game developers when programming networking for multiplayer games.

1.4 Assumptions and limitations

Originally, the project focused primarily on a rich-feature account and game session server however, its scope was largely decreased as greater emphasis was put on the implementation of the UDP protocol. Additionally, the sub-objective of creating a real-time game demo was not completed due to time constraints. Similarly, the depth of the turn-based demo was minimized.

2 Output Summary

Name	udp
Type	Python Package
Size	2093 lines (after formatting)
Credit	
Description	Implimentation of custom udp protocol with client and server
Usage	Base objects for game developer to build on for implimation into their game.
Link	
Name	server
Type	Python Package
Size	700 lines (after formatting)
Credit	
Description	Flask RESTful server responsible for handling comuncation between clients and the database, therefore also responsible for authentication and certificate validation.
Link	Also creates lobbies which include a game server for clients to connect to. Contains the model definitions for use with the database.
Name	client
Type	Python Package
Size	467 lines (after formatting)
Credit	Makes use of a version inputimeout which was modified to disable the automatic appending of new lines on a timeout.
Description	A cmd line client example for end user use. Handles communication to RESTful server via dialog with client. Also responsible for creating game client instances.
Usage	Makes use of inputimeout.
Link	End user client.
Name	rps
Type	Python Package
Size	440 lines (after formatting)
Credit	
Description	A turn-based game demo using the UDP python package. Includes both a client (for use with the client package) and a server (for use as a game server with the server package).
Usage	The package can be slit into two main parts: - The client is used by the client package to communicate with the game server once a game session has been started. - The server is used by the server package when creating a game session.
Link	
Name	tests
Type	Python tests
Size	264 lines (after formatting)
Credit	
Description	A pytest script to test that implimented features are working as expected.
Usage	Used to validate functionality.
Link	

3 Literature Review

3.1 Network Protocols

When considering the transport layer the two primary options for sending data are TCP (Eddy, 2022) and UDP (Postel, 1980). Both have their own strengths and weaknesses.

3.1.1 TCP TCP is a protocol that uses a connection-based approach. It offers a reliable, ordered and error-checked data stream. It is used for a variety of other protocols such as HTTP, FTP and SMTP. These features, while offering benefits also come with drawbacks such as additional overhead which in part contributes to TCP prioritizing data integrity at the expense of latency.

- Reliable

- The sender is notified if a packet is successfully, or unsuccessfully, delivered to its recipient. This means the data is re-sent in the event of packet loss, ensuring that all data is received (unless of a major failure such as the recipient losing connection e.g. though power loss). This, however, incurs a larger overhead than unreliable protocols leading to typically slower data transfer.
 - Ordered
 - Packets are received by their destination in the same order they are sent. This is achieved by assigning a sequence number to each segment of data. The receiver is then able to reassemble the data in the correct order. This, however, can lead to increased latency when a recipient is waiting for a packet after receiving its descendant causing the data stream to hang.
 - Error-checked
 - A checksum is included with the packet data. This allows for a recipient to verify that the data is received in the same state it was sent. In the event of data corruption, the data is re-requested. This also contributes to increased latency as the recipient must wait for the packet to be retransmitted.
-

3.1.2 UDP In contrast, UDP is a connectionless protocol that is unreliable, unordered and provides no error-correction at the interface level (i.e. error-correction must be implemented on the application layer if desired). Despite these simplicities, UDP is arguably more suited to fast, real-time communication where speed is prioritized over integrity.

- Connectionless
 - Due to UDP being a connectionless protocol, UDP is able to broadcast and multicast packets without any additional overhead. This, for example, is useful when a server has to send a game-state update to all game clients.
- Unreliable
 - There are no systems in place to detect if a packet is successfully delivered. This, therefore, means that there is a significant reduction in latency as no resubmission takes place but also means that packets can be lost without either the sender or the recipient being aware.
- Unordered
 - Packets may arrive in any order and it is up to the application to determine the original order. There is no built-in information in the packet to infer the original order either and thus, if this information is desired, must be encoded into the packet payload. This, however, gives the recipient more flexibility, allowing outdated packets to simply be ignored in the event that a newer packet has already been processed.
- No error-checking
 - Though the UDP contains a checksum field, this is not mandatory (at least for IPv4). In the event the checksum is used, any packets that fail the checksum will be dropped at the transport layer and will not reach the application. Due to this, it can be beneficial to not include a checksum in the header and instead implement some form of data validation in the data payload instead.

3.1.3 Comparison When working with time critical data such as that required for real-time video games, particularly those with fast-paced interactions, like FPS such as *Quake* (id Software, 1996) or fighting games such as *Street Fighter IV* (Capcom, 2008) TCP’s overhead leads to a too great latency. Many systems would also prefer to just discard packets in the event of a failure as waiting for retransmission will yield old and outdated information no longer relevant to the current state of the system. These such use-cases are ideal for UDP, though some additional features may have to be implemented on the application level (some borrowed from TCP). The consensus among game developers is typically to implement a custom protocol based on UDP.

“Using TCP is the worst possible mistake you can make when developing a multiplayer game.” *UDP vs. TCP* (Fiedle, 2008b)

Several implementations attempt to add key features to the UDP specification such as:

- Valve’s *GameNetworkingSockets* (ValveSoftware, 2022) allows for a pseudo-connection over UDP as well as allowing reliable and unreliable packets. Though the implementation includes mandatory encryption it lacks any form of compression.
- *ENet* (Salzman, 2024), created for the open-source FPS *Cube* (van Oortmerssen, 2005), provides, solely, reliable UDP packets.

When working with data where latency is not a concern, TCP’s built-in benefits make it a somewhat more suitable choice. For turn-based games like some 4X games such as *Civilization III* (Firaxis Games, 2001) and board games such as *Connect Four* (Howard Wexler, 1974), where latency is less critical, there is argument to be made for either TCP (without *Nagle’s Algorithm*) or UDP. When communicating with a matchmaking or account database, such as through a RESTful server, the benefits of TCP, particularly the added security, far outweigh the potential latency.

3.2 RESTful API

In *Architectural Styles and the Design of Network-based Software Architectures* (Roy Thomas Fielding, 2000) Fielding introduces the REpresentational State Transfer (REST) architectural style. The term RESTful can be used to describe HTTP-based APIs that meet some REST features but this often scrutinizes as an API either adheres to REST or does not. Most uses of the term RESTful actually refer to *HTTP-based Type I* and *HTTP-based Type II* (Jan Algermissen, 2010) where neither adhere to the use of *Hypermedia as the Engine of Application State* defined in REST. The types differ in the use of *Self-Descriptive Messages* i.e. the use of specific media types over generic. General principles state that REST is superior to Type II which in turn is superior to Type I.

“Depending on the degree to which existing media types apply to the problem domain HTTP-based Type II should be considered over HTTP-based Type I because the start-up cost is almost identical. A transition from HTTP-based Type II to REST at a later point in time, however, is rather easy.” *Classification*

of *HTTP-based APIs* (Jan Algermissen, 2010)

Despite this, this document uses the term **RESTful** interchangeably with **HTTP-based Type** due to the communities adoption of the term.

3.3 Security Algorithms

3.3.1 TLS Transport Layer Security (TLS) (Rescorla, 2018) and the similar **Datagram Transport Layer Security (DTLS)** are cryptographic protocols designed to provide secure communication. The protocol describes the data exchanged between the client and server in the handshake. This exchange includes the sharing of an asymmetric (public) key which is used in a key exchange to generate a symmetric session key for use in the rest of communication (i.e. with application data). The **Finished** packet includes a hash of the handshake communications using the session key thus allowing both parties to validate the exchange. The handshake also contains the exchange of certificate(s) allowing parties to validate the identity of the other party.

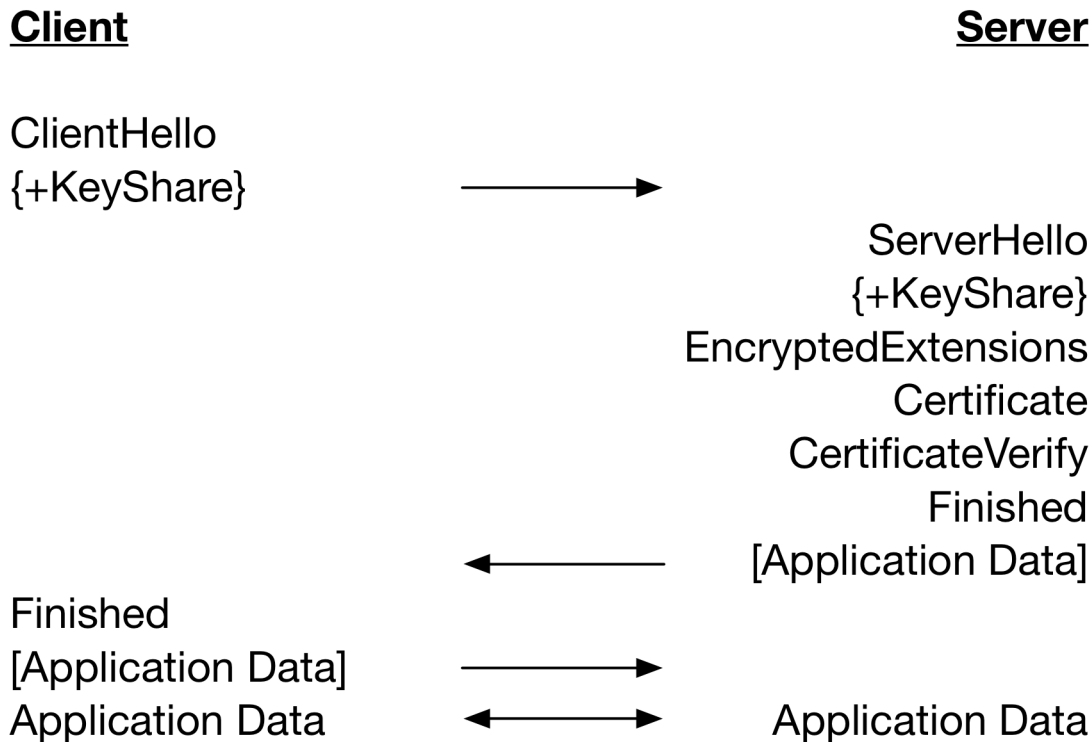


Figure 1: A example of a TLS 1.3 full handshake including a server certificate (wolfSSL, 2019)

3.3.2 Session Keys There are several different options for the asymmetric key used in the key exchange. The primary options (used in TSL 1.3) are either an **Elliptic Curve (EC)**

or Finite Field (FF) which use an Elliptic Curve Diffie-Hellman (ECDH) and Finite Field Diffie-Hellman (FFDH or, more commonly, DH) key exchange respectively. Both are preferred in ephemeral (ECDHE, DHE) form meaning that keys are regenerated for each new session thus meaning the system is less venerable of replay attacks.

The Performance of Elliptic Curve Based Group Diffie-Hellman Protocols for Secure Group Communication over Ad Hoc Networks (Wang, Ramamurthy and Zou, 2006) compares the performance of ECHD against DH and finds that EC outperforms DH in, among other things, both communication time and key generation speed. As such, ECHD(E) is considered to be the preferred method for session key generation.

3.3.3 Authentication The certificate used in the TLS handshake is typically in the form of an X.509 (Internet Engineering Task Force, 2006) containing an identity and a public key which is signed using the respective private key. There are several options for choice in key pair used, with the most common being RSA (Ronald L. Rivest, 1978) and Elliptic Curve Digital Signature Algorithm (ECDSA). DSA, though currently still used, is being phased out largely due to its comparative weakness to other algorithms. ECDSA offers the equivalent level of security to RSA with a smaller key size as well as typically faster encryption and decryption speeds. This can be particularly relevant with a repeated key exchange, but is less relevant in the context of X.509 verification as this process will typically only occur once per session. Historically, RSA has been the de facto choice, but recent years have seen ECDSA grow in adoption. RSA's dominance is largely associated with the algorithm's maturity and existing wide adoption and, for this reason, remains a suitable choice for X.509 signing.

4 Method

4.0 Tools

4.0.1 Programming Language The project is written in Python. This was a language I was most familiar with. The Flask package was used for the RESTful API server in conjunction with SQLAlchemy to communicate with the database.

4.0.2 Database The database language chosen was MySQL. This was deployed in a Docker stack during development for convenience.

4.0.3 IDE Visual Studio (VS) Code was chosen as the primary IDE to write Python. VS Code supports a large variety of different languages via first and third-party plug-ins which was useful when working with some of the additional file types using in this project (e.g. .env, .yaml, .md). Additionally, I was reasonably experienced with creating custom launch.json debug configurations allowing for easy debugging of file in parallel.

4.0.4 Source Control Git and Github were using throughout development for source control management. Branches were frequently used to allow for parallel implementation of different features.

4.1 Methodology

The methodology used thought the project was the **Agile Feature-Driven Development (FDD)** method. This was well suited to the project as it enabled for objectives to be adaptive as a better understanding of the system requirements was gained. Additionally, it allowed for features to be designed, implemented and tested in parallel, ensuring each component was working as expected, before combining into a final cohesive package.

4.2 Analysis

The majority of the analysis can be seen in the *literature review*. Some additional analysis, however, was performed throughout the project as each feature was implemented.

4.3 Design

4.3.1 Packet Specification As stated in the literature review, a custom feature-rich UDP protocol would need to be defined. The additional features include:

- Packet Order
 - There would need to be some way for the recipient to be able to determine the order in which packets were sent thus allowing for old packets to be discarded.
- Reliability
 - There would need to be some way for a sender to be confident that the recipient had received the packet they had sent.
- Error-checking
 - Though UDP provides a built-in checksum, using a custom data validation method would give me both more control as well as the option to still receive corrupted packets on the application layer as the TCP checksum occurs below the application layer.
- Fragmentation
 - Packets should be able to split a packet into fragments. This would be particularly useful when sending a large amount of data via UDP.
- Compression
 - Packets should be able to indicate if a packet's data has been compressed thus allowing decompression to happen automatically. Though compression would be likely impractical with typical packet traffic (likely increasing payload size), packets with large amounts of data such as fragmented packets could be compressed to reduce the number of fragments (thus reducing the number of points of failure).
- Encryption
 - Encryption would provide various benefits such as a recipient being confident in the sender as well as adding security against any attackers. It would also mitigate against packet fabrication. Packets should be able to indicate, in a similar fashion to compression, that they are encrypted so they can be decrypted automatically.

These features were formalized in a Packet Specification document

4.3.2 Database Models The database models were designed using a UML Entity Relationship Diagram (ERD).

4.3.3 API Specification When working with the API, the most logical implementation was to create a RESTful HTTP (TCP) server. Using flask, the web-server could act as a middleman for communication with the database. This allows for data sanitation, easy authorization control and easy scalability.

The TCP server would also be responsible for:

- Matchmaking and joining Lobbies
- Creating Lobbies (and the relevant game servers)
- Managing Accounts
 - Friends
 - Scores
- Certificate Validation

These features were formalize in an API Specification document.

4.4 Implementation

4.4.1 Iteration 1 The first iteration focused on setting up the basis for the custom udp implementation.

4.4.1.1 Packet Specification Implementation Before creating any `Client` or `Server` implementation the packet structure defined in `PACKET_SPEC` was implemented in class definitions with the reliant methods to convert to and from bytes.

4.4.1.2 UDP Node A base class `Node` was created. The `Node` class is responsible for sending and receiving packets.

4.4.1.2.1 Client

A `Client` class was created, inheriting from `Node`. The `Client` class overrides the send methods to use a given `targetAddress`. This means that clients can be created for a specific `Server`

4.4.1.2.1 Server

A `Server` class was created, inheriting from `Node`. The `Server` is initially passive waiting for and replying to incoming packets from a `Client`.

4.4.1.3 DEFAULT Packet The DEFAULT packet sending and receiving was implemented for `Node` using the `packet.Packet` class defined earlier.

4.4.2.1 Threading The `Client` and `Server` are refactored to allow for simultaneous sending and receiving.

Python's (or more specifically CPython's) `Global Interpreter Lock (GIL)` is a mutex that prevents multiple threads from executing Python bytecode at once. This mitigates against race conditions. The GIL is not however a catch all and some actions required additional locking.

4.4.2 Iteration 2 The second iteration focused on expanding the custom UDP implementation with a focus on implementing the authentication and security features outlined in the packet specification.

4.4.1.1 Reliable Flag and ACK Packets The `RELIABLE` flag ensures that packets are delivered. A `Node` will resend a `RELIABLE` packet until it receives acknowledgment through an `ACK` packet.

4.4.2.2 AUTH Packets The `AUTH` packet is used for authenticating a `Node` during the handshake. The `public key` and `certificate` fields defined in the packet specification are implementation agnostic. Ultimately, `Elliptic Curve (EC) Keys` were chosen for use as the key used in the `AUTH` packet. For certificates, and therefore identity verification, `X.509` in conjunction with `RSA` signing is used. The `Node` class has fields for a `X.509` certificate and `EC Private Key` whereas the `RSA` key is defined in the `Client` and `Server`.

4.4.2.3 Handshake The Handshake is loosely defined in the packet specification. As the key chosen for the `AUTH` packet was `EC` an `Elliptic-curve Diffie-Hellman (ECDH)` is used for session key generation.

4.4.2.4 Flags Each flag and its behavior is defined in packet specification. Each flag was implemented such that flag behavior's are automatically performed before sending and after receiving.

4.4.3 Iteration 3 The third iteration focused on finishing implementation of the features outlined in the packet specification.

4.4.3.1 ACK Bits and Rolling Reset The `Node` class was updated to use the available ack bits in the `ACK` packet's headers to provide an additional layer of reliability. Additionally, a rolling reset of the recorded `ACKed` bits was implemented. Without this, upon `sequenceId` wrap around, a `Node` can be misinformed about that bits have been `ACKed`.

4.4.3.2 HEARTBEAT Packets The `HEARTBEAT` packet sending and receiving was implemented using `packet.HeartbeatPacket`. This allows for the `Server` to remove unresponsive `Clients` in a `heartbeatThread`.

4.4.3.3 Callbacks Callbacks were implemented allowing for data to propagate through game `Server` and `Clients` as well allowing for packet data to reach the *application* layer.

4.4.3.4 ERROR Packets The `ERROR` packet sending and receiving was implemented using `packet.ErrorPacket`. Additionally, the errors outlined in the packet specification were implemented as `Exceptions` in `udp.errors`. These `Exceptions` and their relevant handling were put throughout the project.

4.4.3.5 Disconnects The `DisconnectError` is used whenever a `Node` is gracefully terminating. The implementation of the error varies between the `Client` and `Server` with `Client` terminating and the `Server` removing the `Client`.

4.4.4 Iteration 4 The fourth iteration focused on creating the RESTful server and the database models as well as a turn-based game demo. Finally, a end-user `client` was created.

4.4.4.1 DotEnv The `CONSTs` defined across the project were consolidated into a `.env` file. This constance are then loaded at run-time using the `dotenv` package. This provided structure to the project and allowed for easier control over the various variables.

4.4.4.2 Logging Logging using the `logging` model was implemented across the package. The `logger` was set to output to both the console as well as a `paperclip.log` file. This outputs were given different ‘levels’ to avoid cluttering the console output while retaining all generated outputs in the log file.

4.4.4.3 Database Models The database modules defined in the `ERD` were implemented as `SQLAlchemy db.models` allowing the `Server` to initiate the database with the appropriate tables on start-up. Various changes were made between the design and final implementation to match the projects new requirements. These are outlined in the *Results* section.

4.4.4.4 RESTful Server The TCP RESTful API `Server` was implemented using as a `Flask` app. Authentication using `HTTPBasicAuth` is implemented allow for either a username and password or a session key to be used.

The endpoints for the various `API` functionalities are implemented according to the `API` specification.

4.4.4.5 Certificates and Handshake The `udp` handshake is amended to use the RESTful server as an authenticator for certificates. Additionally, the `udp.auth` method to generate certificates is expanded to accept and embed an `Account.id` and an `Account.username` in the certificate fields.

4.4.4.6 RPS Demo A turn-based game demo (Rock, Paper, Scissors) was created containing a game `Server` and `Client`. The `Server` is responsible for evaluating each turn sending the results to the `Clients`. The `Client` is responsible for taking a player input and sending it to the `Server` the `Client` then displays the results received.

4.4.4.7 Client A end-user command-line user-interface `client` was created. The `client` package contains wrappers for communication with `RESTful` server, including authentication. The `client` package then provides a text-based environment for a user for each `API` endpoint. The `client` is also responsible for creating a game `Client` and joining the relevant game `Server`. Finally, the matchmaking logic was implemented to allow for automatic `Lobby` joining when available.

4.5 Tests

The `pytest` module was used to define serval test.

4.6 Reused Code and Tutorials

The `client` package makes use of `inputtimeout` package (Mitsuo Heijo, 2017) to allow for non-blocking inputs. This code was modified to allow prevent the automatic appending of a new line after each timeout.

A large amount of inspiration was taken from the *Reliability and Congestion Avoidance over UDP* (Fiedle, 2008a), in perticualry the use of `ack_bits` in an `ACK` package.

5 Results

5.3 Design

5.3.1 Packet Specification A formal *Packet Specification* was created laying out the different packet types, their flags and flags behaviour and well as various other headers. Additionally, the *Packet Specification* describes a handshake.

The *Packet Specification* is omitted from results section for clarity but is available in full in the documents appendices [Appendices 9.4].

5.3.2 Database Models A ERD was created defining the structure of the various database models and their relationships.

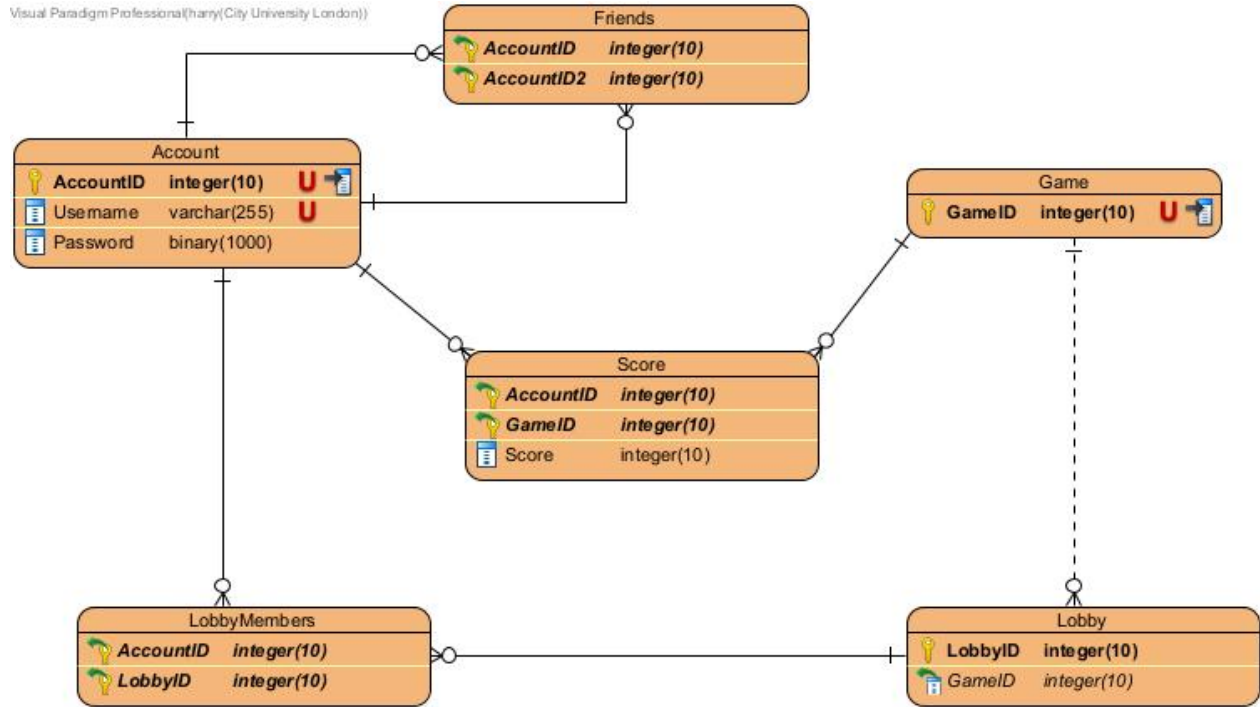


Figure 2: Database Models ERD

5.3.3 API Specification A formal *API Specification* was created describing the various endpoints the RESTful API Server.

The *API Specification* is omitted from results section for clarity but is available in full in the documents appendices [Appendices 9.5].

5.4 Implementation

5.4.1 Iteration 1

5.4.1.1 Packet Specification Implementation Each packet type (defined in the packet spec) is implemented as its own class. All packet classes inherit from a base **Packet** equivalent to the **DEFAULT** packet. The packet classes contain the defined fields as well as static methods to convert from a class instance into bytes (*pack*) and vice versa (*unpack*). The **struct** package allows for converting to and from some integer value into a fixed size bytes with the appropriate padding as well as handling endianness (as UDP uses big-endian). Most class fields are either already integers or can be easily represented as an integer (enum, boolean) but some fields (e.g. public key, certificate, data) require more complex casting. Additionally, the `udp.packet` script includes various **Enums** containing definitions of the **Flags** and packet **Types** and **CONSTs** which define the sizes (in bits) of the headers. These are both used in generation of default (empty) header values as well as a reference in other scripts.

```

1 from enum import Enum
2 class Type(Enum):

```

```

3     DEFAULT = 0
4     ACK = 1
5     AUTH = 2
6     HEARTBEAT = 3
7     ERROR = 4
8
9 class Flag(Enum):
10     RELIABLE = 0
11     CHECKSUM = 1
12     COMPRESSED = 2
13     ENCRYPTED = 3
14     FRAG = 4

```

5.4.1.2 UDP Node Both the `udp.Client` and `udp.Server` classes inherit from a `udp.Node` base class.

5.4.1.2.1 Sending Data

The `Node` class provides methods for sending `Packets` using `socket.socket`. The `sendPacket` method takes an address and a packet instance and dispatches the packed packet to the given host.

```

1 def sendPacket(self, addr:tuple[str, int], p:packet.Packet) -> None:
2     self.socket.sendto(p.pack(p), (addr[0],addr[1]))

```

The `sendPacket` method is typically not directly called, with relevant send methods existing for each packet type. As `Node` is responsible for keeping an internal `sequenceId`, is able to set each packet and then increment its record.

```

1 def sendDefault(self, addr:tuple[str, int], data:bytes|None=None) -> None:
2     p = packet.Packet(sequence_id=self.sequenceId, data=data)
3     self.sequenceId += 1
4     self.sendPacket(addr, p)

```

5.4.1.2.2 Receiving Data

The `Node` class also provides a method for receiving packets from the `socket`. This allows for packets to be packed into an instance before they are returned.

```

1 def receivePacket(self) -> tuple[packet.Packet, tuple[str, int]]:
2     data, addr = self.socket.recvfrom(BUFFER_SIZE)
3     p = packet.unpack(data)
4     return p, addr

```

5.4.1.2.3 UDP Client

The `Client` also includes a target address and overrides the `Node`'s send methods to set the destination to be its target address. The `addr` field is still included in the method so that function calls from `Node` do not break.

```
1 def sendDefault(self, addr:tuple[str,int]=None, data:bytes|None=None):
2     return super().queueDefault(self.targetAddr, data=data)
```

5.4.1.2.4 UDP Server

The `Server` is initially passive, only replying to incoming packets from a client.

```
1 def mainloop(self):
2     while True:
3         p, addr = self.receivePacket()
4         # logic to process and reply (if needed)
5         # e.g. self.sendDefault(addr, data=b"Hello Client")
```

5.4.1.3 DEFAULT Packet The `DEFAULT` packet takes a list of booleans flags in addition to a data field. The flags field defaults to any list of `False` if no flags are specified.

```
1 def sendDefault(self, addr:tuple[str, int], flags: list[bool] = [0 for _ in
    range(packet.FLAGS_SIZE)], data:bytes|None=None) -> None:
2     p = packet.Packet(sequence_id=self.sequenceId, flags=flags, data=data)
3     self.sequenceId += 1
4     self.sendPacket(addr, p)

1 def receiveDefault(p: packet.Packet, addr: tuple[str, int]):
2     pass
```

5.4.1.4 Threading In order to be able to send and receive packets simultaneously both actions are contained in a `threading.Thread`.

5.4.1.4.1 Thread Safety

The GIL does not protect against such interaction as the `+=` operator. As such the `sequenceId` variable must be incremented using a `threading.Lock` so that all threads can increment the `sequenceId` safely.

```
1 def incrementSequenceId(self) -> None:
2     with self.sequenceIdLock:
3         self.sequenceId += 1
```

The `threading` module also provides the `Event` class. This allows easy communication between threads and is used for the `isRunning` field to stop all threads whenever any thread resets the `Event` to `False`.

5.4.1.4.2 Inbound Thread

The `inboundThread` field is defined as `Thread(name="Inbound", target=self.listen, daemon=True)` on a `Node`'s `__init__`.

The `listen` method waits for an incoming package and yield to the `receive` method. This happens in a loop until `isRunning` is reset.

```
1 def listen(self):
2     while self.isRunning.is_set():
3         p, addr = self.receivePacket()
4         self.receive(p, addr)
```

The `receive` method is responsible for passing the package to the appropriate packet type receive method.

```
1 def receive(self, p: packet.Packet, addr: tuple[str, int]):
2     if p is not None:
3         match (p.packet_type):
4             case packet.Type.DEFAULT:
5                 return self.receiveDefault(p, addr)
6                 # other packet type cases omitted for clarity
7             case _:
8                 raise TypeError(f"Unknown packet type '{p.packet_type}' for packet {p}")
```

The `Server` uses its own `listen` method. It uses this to only allow certain packets depending on the state of the client's handshake. If the client has not yet initiated handshake, and thus does not exist, all packets other than `AUTH` are dropped. If a client has started, and thus exists, but has not completed the handshake only `AUTH` and `ACK` packets are passed. The `Server` otherwise accepts all packets from a *connected* client (i.e. a client with a completed handshake).

```
1 def listen(self) -> None:
2     while self.isRunning.is_set():
3         p, addr = self.receivePacket()
4         if p is not None and addr is not None:
5             if self.checkClientExists(addr): # client exists
6                 if self.getHandshake(addr): # client handshake complete =>
7                     allow all packet types
8                     self.receive(p, addr)
9                 else:
10                    if p.packet_type in (packet.Type.AUTH, packet.Type.ACK):
11                        # client handshake incomplete => drop all non-AUTH /
12                        non-ACK packets
13                        self.receive(p, addr)
14            else:
```

```

12         if p.packet_type in (packet.Type.AUTH): # client not exists
13             => drop all non-AUTH packets
14             self.receive(p, addr)

```

5.4.1.4.3 Outbound Thread

The `outboundThread` field is defined as `Thread(name=f"Outbound", target=self.sendQueue, daemon=True)` on a `Node`'s `__init__`. In order for the `sendQueue` method to be able to send packages they first need to be added to a `queue.Queue`. A `Queue` is a thread-safe data structure with built in locking, allowing for multiple threads to safely add and remove data in the same variable.

```

1 def sendQueue(self):
2     while self.isRunning.is_set():
3         addr, p = self.queue.get()
4         self.sendPacket(addr, p)
5         self.queue.task_done()
6         time.sleep(SLEEP_TIME) # some small time delay

```

The `send` methods are replaced by their receptive queue methods. Instead of sending the packet they instead yield to the `queuePacket` method.

```

1 def queueDefault(self, addr:tuple[str, int], data:bytes|None=None) -> None:
2     p = packet.Packet(sequence_id=self.sequenceId, data=data)
3     self.incrementSequenceId()
4     self.queuePacket(addr, p)

```

The `queuePacket` method, in turn, appends the packet (with the relented destination address) to the queue. This method is also used to apply the `reliant` flag behavior(s).

```

1 def queuePacket(self, addr: tuple[str, int], p: packet.Packet) -> None:
2     # logic for flags omitted
3     self.queue.put((addr, p))

```

5.4.1.4.4 Server Clients

The `Server`, now being threaded, is able to accept multiple clients. Whenever a new handshake is started by a client, a new `Node` is created and added to a dictionary field `clients` (using the client address as the key). The `Node` uses the server's socket to send replies to a client and as such the `Node` class is refactored to take a `socket` as well as a `Lock`. The `Lock` is used whenever a packet is sent, to ensure thread-safety. Using a `Node` for tracking clients allows for each client connection to have its own `sequenceId`, (as well as `sessionKey`, `ecKey`, etc. described in later iterations).

```

1 def makeClient(self, clientAddr: tuple[str, int]) -> None:
2     c = node.Node(
3         clientAddr,
4         sendLock=self.sendLock,
5         socket=self.socket,

```

```

6         )
7         c.outboundThread.start()
8         with self.clientsLock:
9             self.clients[clientAddr] = c

```

Additionally, all `Node` fields are refactored to use getter and setters taking an `addr`. This allows the `Server` class to override the setter and getters to instead return the relevant field from client in the dictionary. The `Server` also uses a `Lock` when retrieving client attributes.

```

1 def getSequenceId(self, clientAddr: tuple[str, int]) -> int | None:
2     with self.clientsLock:
3         return (
4             self.clients[clientAddr].sequenceId
5             if clientAddr in self.clients
6             else None
7         )

```

5.4.2 Iteration 2

5.4.2.1 RELIABLE Flag and ACK Packets 5.4.2.1.1 Sending a RELIABLE packet

When queuing a RELIABLE packet the `Node` sets the relevant `sentAckBit` to false before adding to the send queue.

```

1 def queuePacket(self, addr: tuple[str, int], p: packet.Packet) -> None:
2     if p.flags[packet.Flag.RELIABLE.value]:
3         self.setSentAckBit(addr, p.sequence_id, False) # set relevant ack bit
4         # to False
5         self.queue.put((addr, p))

```

After a `Node` sends a packet with the RELIABLE flag set it appends the packet back to the end of the queue. The next time the `Node` goes to send the packet it first checks against its record of received ACKed packets. If the packet has already been ACK, the recipient has given confirmation of receipt and the packet does not need to be resent. This helps to mitigate against packet loss as *critical* packets which are marked as RELIABLE will be resent until the `Node` is confident that the recipient has received it.

```

1 def sendQueue(self):
2     while self.isRunning.is_set():
3         addr, p = self.queue.get()
4         if p.flags[packet.Flag.RELIABLE.value]:
5             if self.getSentAckBit(addr, p): # checks if ACKed
6                 self.queue.task_done()
7                 continue # skips
8             else:
9                 self.sendPacket(addr, p) # sends
10                self.queue.task_done()

```

```

11         self.queue.put((addr, p)) # re-adds to the queue
12     else:
13         self.sendPacket(addr, p)
14         self.queue.task_done()
15         time.sleep(SEND_SLEEP_TIME)

```

5.4.2.1.2 Receiving a RELIABLE packet

When a Node receives a packet with the RELIABLE flag set, in addition to processing the packet as normal, the Node appends an ACK packets to its queue. The ACK package's ACK ID is set to the Sequence ID of the incoming package. The Node also keeps a record of sent ACK packet's to ensure that any repeat packets do not propagate to the *application layer*.

```

1 def handleReliable(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
2     if p.flags[packet.Flag.RELIABLE.value]:
3         self.setRecvAckBit(addr, p.sequence_id, True) # set relevant recv
4                                                         bit
5         self.queueACK(addr, p.sequence_id) # queues and ACK
6         return True
7     else:
8         return False

```

The `handleReliable` method is called by the `handleFlags` method. This method is responsible for processing all flags *before* the Node attempts to process the packet instance.

```

1 def handleFlags(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
2     self.handleReliable(p, addr)
3     return True

```

As such, the `receive` method is modified to first handle flags before processing.

```

1 def receive(self, p: packet.Packet, addr: tuple[str, int]):
2     if p is not None:
3         if self.handleFlags(p, addr):
4             match (p.packet_type):
5                 # packet type cases omitted for clarity
6                 case _:
7                     raise TypeError(f"Unknown packet type '{p.packet_type}'
8                                     for packet {p}")

```

5.4.2.1.3 Sending an ACK Packet

In addition to all of the fields used to queue a DEFAULT packet, the ACK packet also takes an `ackId` representing the packet to which the ACK is acknowledging.

```

1 def queueACK(self, addr: tuple[str, int], ackId: int, flags: list[bool] = [0
2     for _ in range(packet.FLAGS_SIZE)], data: bytes | None = None) -> None:
3     p = packet.AckPacket(
4         sequence_id=self.getSequenceId(addr),

```

```

4         flags=flags,
5         ack_id=ackId,
6         data=data,
7     )
8     self.incrementSequenceId(addr)
9     self.queuePacket(addr, p)

```

5.4.2.1.4 Receiving an ACK Packet

When a `Node` receives an ACK packet it sets the relevant ACK ID in its record of received ACKed packets to `true`, thus preventing resending a confirmed packet.

```

1 def receiveAck(self, p: packet.AckPacket, addr: tuple[str, int]) ->
   tuple[packet.Packet, tuple[str, int]]:
2     self.setSentAckBit(addr, p.ack_id, True)
3     return (p, addr)

```

5.4.2.2 AUTH Packets The X.509 certificates are generated in `udp.auth` taking a RSA private key for signing and are self-signed (i.e the subject is also the issuer).

```

1 def generateUserCertificate(key) -> x509.Certificate:
2     name = [
3         x509.NameAttribute(NameOID.ORGANIZATION_NAME, ORG_NAME), # ORG_NAME
   defined as const e.g. "Paperclip"
4         x509.NameAttribute(NameOID.COMMON_NAME, COMMON_NAME), # COMMON_NAME
   defined as const e.g. "127.0.0.1"
5     ]
6     subject = issuer = x509.Name(name) # self signed
7     cert = (
8         x509.CertificateBuilder()
9         .subject_name(subject)
10        .issuer_name(issuer)
11        .public_key(key.public_key())
12        .serial_number(x509.random_serial_number())
13        .not_valid_before(datetime.datetime.now(datetime.timezone.utc))
14        .not_valid_after(
15            datetime.datetime.now(datetime.timezone.utc) +
16            datetime.timedelta(days=1) # valid for one day
17        )
18        .add_extension(
19            x509.SubjectAlternativeName([x509.DNSName("localhost")]), # self
   signed
20            critical=False,
21        ).sign(key, hashes.SHA256())

```

```

22     )
23     return cert

```

The key and certificate are converted to and from DER bytes format when packing and unpacking.

```

1 def getDerFromPublicEc(publicKey: ec.EllipticCurvePublicKey) -> bytes:
2     ecDer = publicKey.public_bytes(
3         encoding=serialization.Encoding.DER,
4         format=serialization.PublicFormat.SubjectPublicKeyInfo,
5     )
6     return ecDer

1 def getPublicEcFromDer(publicKeyDer: bytes) -> ec.EllipticCurvePublicKey:
2     ec_ = serialization.load_der_public_key(publicKeyDer)
3     return ec_

```

5.4.2.2.1 Sending an AUTH Packet

The `queueAuth` packet takes the additional fields of a certificate and a public key.

```

1 def queueAuth(self, addr: tuple[str, int], cert: Certificate, publicEc:
2     auth.ec.EllipticCurvePublicKey) -> None:
3     p = packet.AuthPacket(
4         sequence_id=self.getSequenceId(addr), certificate=cert,
5         public_key=publicEc
6     )
7     self.incrementSequenceId(addr)
8     self.queuePacket(addr, p)

```

5.4.2.2.2 Receiving an AUTH Packet

The base `Node` class contains a `receiveAuth` method exclusively for use in overriding.

```

1 def receiveAuth(self, p: packet.AuthPacket, addr: tuple[str, int]) ->
2     tuple[packet.Packet, tuple[str, int]]:
3     raise NotImplementedError(
4         "Node should not receive auth. A child class must overriding."
5     )

```

The `Server` overrides this method with the logic for handling a handshake. The `Client` class, however, does not make use of this method as it handles all AUTH packets during its `connect` method.

5.4.2.3 Handshake The handshake is implemented according to the packet specification. The session key is generated with a ECDH key exchange in `udp.auth`.

```
1 def generateSessionKey(localKey: ec.EllipticCurvePrivateKey, peerKey:
    ec.EllipticCurvePublicKey) -> bytes:
2     sessionSecret = localKey.exchange(ec.ECDH(), peerKey)
3     sessionKey = HKDF(
4         algorithm=hashes.SHA256(), length=32, salt=None, info=b"handshake
        data"
5     ).derive(sessionSecret)
6     return sessionKey
```

The `Finished` is computed by calculating the HMAC of the `finishedLabel` and messages using the session key.

```
1 def generateFinished(sessionKey: bytes, finishedLabel: bytes, messages:
    bytes):
2     hashValue = hashes.Hash(hashes.SHA256())
3     hashValue.update(messages)
4     hashValue = hashValue.finalize()
5     prf = hmac.HMAC(sessionKey, hashes.SHA256())
6     prf.update(finishedLabel)
7     prf.update(hashValue)
8     prf = prf.finalize()
9     return prf
```

5.4.2.3.1 Client Handshake

The `Client` is responsible for starting the handshake using the `connect` method. It starts by starting the `outboundThread` so it is able to send packets. It is then able to send a `AUTH` packet. The client then waits to receive both the `AUTH` and `ACK` packet from the `Server`.

When the `AUTH` packet is received the `Client` first generates the session key. It is then able to compute the `Finished` which is sent as the data field of an `ACK` packet. It also checks the validity of the `Server`'s certificate, aborting the connection attempt on a failure.

When both the `ACK` and `AUTH` packets are received the `Client` checks the validity of the `Finished` by checking its version of `Finished` against the contents of the `ACK` packet. On a failure, the connection is aborted. On a success, the `Client` starts the `inboundThread` and the connection is considered complete.

```
1 def connect(self) -> None:
2     self.outboundThread.start() # start outbound
3     self.queueAuth(self.targetAddr, self.cert, self.ecKey.public_key()) #
        send auth
4     authPacket = None
5     ackPacket = None
6     while True:
```

```

7     p, addr = self.receivePacket()
8     if p is not None:
9         # logic
10        if p.packet_type == packet.Type.AUTH: # AUTH packet -> generate
            session key, validate certificate, queueFinished
11            authPacket = p
12            self.sessionKey = auth.generateSessionKey(
13                self.ecKey, p.public_key
14            )
15            if not self.validateCertificate(p.certificate):
16                # certificate not valid
17                # abort
18                break
19            self.queueFinished(
20                self.targetAddr, p.sequence_id, self.sessionKey
21            )
22        elif p.packet_type == packet.Type.ACK: # ACK packet
23            ackPacket = p
24            self.receiveAck(p, addr)
25            if authPacket is not None and ackPacket is not None: # wait until
                both parts received
26                break
27        else:
28            # Server not responsive
29            # abort
30            break
31    if self.validateHandshake(ackPacket.data): # check finished
32        # success
33        self.inboundThread.start() # start inbound
34    else:
35        # abort

```

5.4.2.3.2 Server Handshake

The **Server**, being a passive listener to the handshake, overrides `receiveAuth` to respond accordingly. The handshake logic varies slightly depending on if the client is a new or existing client (i.e. reconnecting).

If the client is new, the **Server** first ensures it has space (set by the `maxClients` field) and then creates a new client.

If a new client has been created or the client already exists, the **Server** first checks the validity of the client's certificate. The **Server** then regenerates the **Node's** `ecKey` to be used in generating the `sessionKey`. It is then able to send both the reply **AUTH** and **ACK** (containing the generated **Finished**).


```

1 def receiveAuth(self, p: packet.AuthPacket, addr: tuple[str, int]) ->
  tuple[packet.AuthPacket, tuple[str, int]]:
2     if addr not in self.clients: # new client
3         if self.isNotFull(): # check space
4             valid, accountId = self.validateCertificate(p.certificate)
5             if not valid:
6                 # invalid certificate
7                 # abort
8                 return
9             else:
10                self.makeClient(addr, p.certificate, accountId)
11                self.regenerateEcKey(addr)
12                sessionKey = auth.generateSessionKey(
13                    self.getEcKey(addr), p.public_key
14                )
15                self.setSessionKey(addr, sessionKey) # sets client sessionKey
16                for later reference
17                self.queueAuth(addr, self.cert,
18                    self.getEcKey(addr).public_key())
19                self.queueFinished(addr, p.sequence_id,
20                    self.getSessionKey(addr))
21            else:
22                # no space
23                # abort
24                return
25        else:
26            sessionKey = auth.generateSessionKey(self.getEcKey(addr),
27                p.public_key)
28            if addr in self.clients: # existing client
29                if self.getSessionKey(addr) != sessionKey: # new client sessionKey
30                    valid, accountId = self.validateCertificate(p.certificate)
31                    if not valid:
32                        # invalid certificate
33                        # abort
34                        # remove client
35                        return
36                    else:
37                        self.regenerateEcKey(addr)
38                        sessionKey = auth.generateSessionKey(
39                            self.getEcKey(addr), p.public_key
40                        )
41                        self.setSessionKey(addr, sessionKey) # make new session key
42                        self.queueAuth(addr, self.cert,
43                            self.getEcKey(addr).public_key())
44                        self.queueFinished(addr, p.sequence_id,

```

```

40         self.getSessionKey(addr))
    return (p, addr)

```

When the **Server** receives an ACK packet the server it checks that the packet's data matches the generated Finished. If the check fails, the connection is aborted and the handshake is not set to complete.

```

1 def receiveAck(self, p: packet.AckPacket, addr: tuple[str, int]) -> None:
2     super().receiveAck(p, addr)
3     if p.data is not None and not self.getHandshake(addr): # ack has
4         # payload & client has not completed handshake => validate handshake
5         if not self.validateHandshake(addr, p.data): # checks and sets
6             the clients handshake
7             # invalid finish
8             # abort
9             return
10        else:
11            # success
12            pass

```

5.4.2.4 Flags All flags behaviors are executed on a packet (where set) before sending and after receiving meaning that the data yielded to the *application* layer is as it was originally set.

5.4.2.4.1 ENCRYPT

Encryption and decryption is performed using AES with the session key and a 16-bit init vector.

```

1 def generateCipher(sessionKey: bytes, iv: bytes = generateInitVector()) ->
2     tuple[Cipher, bytes]:
3     cipher = Cipher(algorithms.AES(sessionKey), modes.CBC(iv))
4     return cipher, iv

```

5.4.2.4.1.1 Encryption

When a **Node** goes to queue a packet with the ENCRYPT flag set it calls `p.encryptData(self.getSessionKey)` (where `p` is the packet). The `encryptData` method generates an `init` vector and subsequent `cipher` before performing the encryption on the data.

```

1 def encryptData(self, session_key: bytes) -> None:
2     self.flags[Flag.ENCRYPTED.value] = 1 # ensure flag set
3     iv = (
4         self.init_vector
5         if self.init_vector is not None
6         else auth.generateInitVector() # equivalent to os.urandom(16)

```

```

7     )
8     cipher, iv = auth.generateCipher(session_key, iv)
9     self.init_vector = iv # assign to header
10    self.data = auth.encryptBytes(cipher, self.data)

```

The `encryptBytes` method includes the `autoPad` boolean. This ensure that the `rawBytes` are a suitable length for the cipher to encrypt.

```

1 def encryptBytes(cipher: Cipher, rawBytes: bytes, autoPad=True) -> bytes:
2     if autoPad:
3         padder = padding.PKCS7(algorithms.AES.block_size).padder()
4         rawBytes = padder.update(rawBytes) + padder.finalize()
5     encryptor = cipher.encryptor()
6     encryptedBytes = encryptor.update(rawBytes) + encryptor.finalize()
7     return encryptedBytes

```

5.4.2.4.1.2 Decryption

When a `Node` receives a packet with the `ENCRYPT` flag set, it calls `p.decryptData(self.getSessionKey(add` (where `p` is the packet). The `decryptData` method first checks that the packet is flagged appropriately (to prevent trying to decrypt an unencrypted packet). It then generates a `cipher` using the packet's init vector and uses this to decrypt the packet data.

```

1 def decryptData(self, session_key: bytes) -> None:
2     if self.flags[Flag.ENCRYPTED.value]:
3         cipher = auth.generateCipher(session_key, self.init_vector)[0]
4         self.data = auth.decryptBytes(cipher, self.data)
5     else:
6         # not flagged for decryption

```

The `decryptBytes` method contains the `autoUnpad` boolean. This is used to automatically remove any padding left by the encryption process.

```

1 def decryptBytes(cipher: Cipher, encryptedBytes: bytes, autoUnpad: bool =
    True) -> bytes:
2     decryptor = cipher.decryptor()
3     decryptedBytes = decryptor.update(encryptedBytes) + decryptor.finalize()
4     if autoUnpad:
5         unpadder = padding.PKCS7(algorithms.AES.block_size).unpadder()
6         decryptedBytes = unpadder.update(decryptedBytes) + unpadder.finalize()
7     return decryptedBytes

```

5.4.2.4.2 COMPRESS

The `COMPRESSED` flag allows for the data to be automatically compressed and decompressed when the flag is set.

5.4.2.4.2.1 Compression

When the Node goes to queue a packet with the COMPRESSED flag set it first calls for the packet to be compressed using the packet's `compressData` method.

```
1 def compressData(self) -> None:
2     self.flags[Flag.COMPRESSED.value] = 1 # ensure flag set
3     self.data = utils.compressData(self.data)
```

The `utils.compressData` method uses the `zlib` library with the default `level`, which compromises speed with efficiency, but the negative of the default `wbits` to ensure that no header or checksum is appended to the bytes as this would create unnecessary overhead.

```
1 def compressData(data: bytes) -> bytes:
2     # default speed
3     # no header or checksum
4     return zlib.compress(data, -1, -15)
```

5.4.2.4.2 Decompression

When a Node receives a packet with the COMPRESS flag set it first calls for the packet to be decompressed using the packet's `decompressData` method.

```
1 def decompressData(self) -> None:
2     if self.flags[Flag.COMPRESSED.value]:
3         self.data = utils.decompressData(self.data)
4     else:
5         # not flagged for decompression
```

The `utils.decompressData` method performs the `zlib` decompression using the same `wbits` as the compression to not expect a header or checksum.

```
1 def decompressData(data: bytes) -> bytes:
2     # no header or checksum
3     return zlib.decompress(data, -15)
```

5.4.2.4.3 CHECKSUM

The checksum is defined in the packet specification as a CRC-32 checksum of a packet's data. The `zlib` library includes a method to generate a CRC-32 checksum, which this project utilizes.

```
1 def generateChecksum(data: bytes) -> int:
2     return zlib.crc32(data)
```

5.4.2.4.3.1 Setting a Checksum

When a Node goes to queue a packet with the CHECKSUM flag set, it first calls for the checksum to be set using the packet's `setChecksum` method.

```
1 def setChecksum(self) -> None:
2     self.flags[Flag.CHECKSUM.value] = 1 # ensure flag set
```

```

3     data = self.data if self.data is not None else b"" # sets to empty byte
      string if None
4     self.checksum = utils.generateChecksum(data) # assign to header

```

5.4.2.4.3.2 Validating a Checksum

When a Node receives a packet with the CHECKSUM flag set, it first checks the packet's data against the checksum using the packet's `validateChecksum` method. The Node does not drop the packet on a failure but does raise a warning that the checksum failed.

```

1 def validateChecksum(self) -> bool:
2     if self.flags[Flag.CHECKSUM.value]:
3         data = self.data if self.data is not None else b"" # sets to empty
          byte string if None
4         return self.checksum == utils.generateChecksum(data)
5     else:
6         # not flagged for checksum validation

```

5.4.2.4.4 FRAG

The FRAG flag allows for the automatic *fragmentation* of the packet's data into several sub-packages. These are then reassembled into a final *super-packet* once the recipient has collected all the fragments.

5.4.2.4.4.1 Fragmentation

When the Node goes to queue a packet with the FRAG flag set, the Node first calls the packet's `fragment` method. This method splits the packet's data into fragmented chunks and creates a list of *fragment* packets.

```

1 def fragment(self):
2     self.flags[Flag.FRAG.value] = 1 # ensure flag set
3     header = Packet._getHeader(self) # returns dictionary of packet's headers
      (where set)
4     fragData = utils.fragmentData(self.data)
5     fragment_number = len(fragData)
6     return [
7         self._createFragment(
8             header, fragment_id=i, fragment_number=fragment_number, data=data
          # set fragment_id, fragment_number and data through
          comprehension
9         )
10    for i, data in enumerate(fragData)
11    ]

```

The `_createFragment` classmethod creates a new class instance with the given attributes.

```

1 @classmethod
2 def _createFragment(
3     cls, header: dict, fragment_id: int, fragment_number: int, data: bytes
4 ):
5     return cls(
6         **header,
7         fragment_id=fragment_id,
8         fragment_number=fragment_number,
9         data=data,
10    )

```

The `utils.fragmentData` method splits the data into a list of bytes, splitting the data into fragments with a max size `MAX_FRAGMENT_SIZE`. The `MAX_FRAGMENT_SIZE` is set to 988 to keep the total packet size under 1024 (`SOCKET_BUFFER_SIZE`) when including the maximum theoretical header size.

```

1 def fragmentData(data: bytes) -> list[bytes]:
2     return [
3         data[i : i + MAX_FRAGMENT_SIZE] for i in range(0, len(data),
4             MAX_FRAGMENT_SIZE)
5     ]

```

5.4.2.4.4.2 Defragmentation

In order to collect all the fragments for reassembly, the `Node` class contains a dictionary `fragBuffer` using the fragments `sequence_id` as the key and a list of the fragments as the values. When a `Node` receives a packet with the `FRAG` flag set it appends it to the `fragBuffer` (creating a new entry if required). It then checks to see if all the `fragBuffer[p.sequence_id]` are set. If so, the fragments can be recompiled into the *super-packet* and passed to `receive` and the buffer entry can be deleted.

```

1 def handleFrag(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
2     if p.flags[packet.Flag.FRAG.value]:
3         if p.sequence_id not in self.getFragBuffer(addr): # new fragment
4             sequence id
5             self.getFragBuffer(addr)[p.sequence_id] = [
6                 None for _ in range(p.fragment_number) # Empty list with size
7                 == p.fragment_number
8             ]
9         self.getFragBuffer(addr)[p.sequence_id][p.fragment_id] = p
10        if all(self.getFragBuffer(addr)[p.sequence_id]): # all list members
11            not None
12            defrag = p.defragment(self.getFragBuffer(addr)[p.sequence_id])
13            del self.getFragBuffer(addr)[p.sequence_id] # remove fragment
14            sequence id from dict
15            self.receive(defrag, addr)
16        return True

```

```

13     else:
14         return False

```

The `defragment` classmethod creates a new *super-packet* from a list of fragments.

```

1 @classmethod
2 def defragment(cls, frags):
3     if frags[0].flags[Flag.FRAG.value]: # assumes all packets flag state
4         based on the first's
5         header = Packet._getHeader(frags[0])
6         header["flags"][Flag.FRAG.value] = 0 # de-sets the FRAG flag
7         data = utils.defragmentData([frag.data for frag in frags])
8         return cls(**header, data=data)
9     else:
10         # not flagged for defragmentation

```

The `utils.defragmentData` method takes a list of bytes and returns the joined cohesive bytes.

```

1 def defragmentData(fragments: list[bytes]) -> bytes:
2     return b"".join(fragments)

```

5.4.2.4.5 Automatic Handling

The Node's `queuePacket` method is now able to handle all flag variants. The order in which the Node performs each flag action is based on the order described by the `Flags`.

```

1 def queuePacket(self, addr: tuple[str, int], p: packet.Packet) -> None:
2     # reliable -> checksum -> compress -> encrypt -> frag
3     if p.flags[packet.Flag.RELIABLE.value]:
4         self.setSentAckBit(addr, p.sequence_id, False)
5     if p.flags[packet.Flag.CHECKSUM.value]:
6         p.setChecksum()
7     if p.flags[packet.Flag.COMPRESSED.value]:
8         p.compressData()
9     if p.flags[packet.Flag.ENCRYPTED.value]:
10         p.encryptData(self.getSessionKey(addr))
11     if p.flags[packet.Flag.FRAG.value]:
12         frags = p.fragment()
13         for frag in frags:
14             self.getQueue(addr).put((addr, frag)) # queue each fragment
15     else:
16         self.getQueue(addr).put((addr, p)) # queue packet

```

Similarly, the Nodes `handleFlags` method is now able to handle all flag variants. The order in which the Node handles each flag is based on the **reverse** of the order described by the

Flags. All the handle methods return a boolean indicating if the flag is present and, thus, the flag action was performed. This is used to return a boolean based on if the packet was a fragment packet. The `receive` method checks the result of `handleFlags` and skips further processing in the event that the flag was a fragment.

```

1 def handleFlags(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
2     # defrag -> decrypt -> decompress -> validate checksum -> reliable
3     if self.handleFrag(p, addr):
4         return False
5     else:
6         self.handleEncrypted(p, addr)
7         self.handleCompressed(p, addr)
8         self.handleChecksum(p, addr)
9         self.handleReliable(p, addr)
10    return True

```

5.4.3 Iteration 3

5.4.3.1 ACK Bits and Rolling Reset The `Node` class utilizes its local record of sent ACKed to set the ACK Bits. This helps to mitigate against packet loss as each ACK packet also includes an acknowledgment of the last 16 packets (if received). This means when a Node receives an ACK packet as well as setting the ACK ID in its received ACKed packets it also iterates over all the bits in the ACK Bits (with their ID set according to the packet specification) and sets accordingly.

```

1 def receiveAck(self, p: packet.AckPacket, addr: tuple[str, int]) ->
    tuple[packet.Packet, tuple[str, int]]:
2     self.setNewestSeqId(
3         addr, self.getNewerSeqId(self.getNewestSeqId(addr), p.sequence_id)
4     )
5     self.setSentAckBit(addr, p.ack_id, True)
6     # set all bits from ack bits to true (to mitigate lost ack)
7     for i, j in enumerate(range(p.ack_id - 1, p.ack_id - 1 -
        packet.ACK_BITS_SIZE, -1)):
8         if p.ack_bits[i]:
9             self.setSentAckBit(addr, j, True)
10    return (p, addr)

```

The `Node` class also implements a rolling reset on its record of sent ACKs. Without this, the record becomes incorrect after the `sequence id` wrap around at 2^{16} . To do this the Node keeps a record of the *newest* sequence id it has received. To calculate the newer of two ids both ids are subtracted from each other to create two difference values which are both modded with 2^{16} . The smallest difference gives the newer id.

```

1 def getNewerSeqId(currentSeqId: int, newSeqId: int) -> int:
2     currentDiff = (newSeqId - currentSeqId) % (2**16)

```



```

3     newDiff = (currentSeqId - newSeqId) % (2**16)
4     if newDiff < currentDiff:
5         return currentSeqId
6     else:
7         return newSeqId

```

Every time a packet is received, it is checked against the newest sequence id and the newest id is updated accordingly. Then, when a RELIABLE packet is received, after updating the newest sequence id it calls `resetBits`.

```

1 def handleReliable(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
2     if p.flags[packet.Flag.RELIABLE.value]:
3         self.setNewestSeqId(
4             addr, self.getNewerSeqId(self.getNewestSeqId(addr), p.sequence_id)
5         )
6         self.setRecvAckBit(addr, p.sequence_id, True)
7         self.resetRecvAckBits(addr)
8         self.queueACK(addr, p.sequence_id)
9         return True
10    else:
11        return False

```

The `resetBits` method iterates its sent ACKs from the newest sequence id to (newest sequence id + half of array)% 2**16 and resets the bits to `None`. This resets half of all bits after the newest sequence id, accounting for the wrap around, to ensure that there is never confusion from a previously ACKed packet from before a wrap around.

```

1 def resetBits(sentACKs: list[bool | None]) -> None:
2     ACK_RESET_SIZE = 2**15 # 2**16 / 2
3     end = (newestSeqId - ACK_RESET_SIZE) % 2**16
4     counter = 0
5     while counter != end:
6         sentACKs[(newestSeqId + 1 + counter) % 2**16] = None
7         counter += 1

```

5.4.3.2 HEARTBEAT Packets When a Node receives a packet it updates its `heartbeat` field to be the current datetime (`datetime.datetime.now()`). When a Server receives a packet from a client it also updates its heartbeat record for that client.

The `queueHeartbeat` method takes the additional boolean `heartbeat`.

```

1 def queueHeartbeat(self, addr: tuple[str, int], heartbeat: bool, flags:
2     list[bool] = [0 for _ in range(packet.FLAGS_SIZE)], data: bytes | None =
    None) -> None:
3     p = packet.HeartbeatPacket(

```

```

3         sequence_id=self.getSequenceId(addr),
4         flags=flags,
5         heartbeat=heartbeat,
6         data=data,
7     )
8     self.incrementSequenceId(addr)
9     self.queuePacket(addr, p)

```

The heartbeatThread uses the heartbeat method.

```

1 def startThreads(self) -> None:
2     super().startThreads()
3     self.heartbeatThread.start()

```

The Server checks every HEARTBEAT_MIN_TIME (30 seconds) each *connected* client's heartbeat delta (now() - client.heartbeat). If the heartbeat delta is greater than some HEARTBEAT_MAX_TIME (120 seconds) the client is dropped as it can be assumed to have either terminated or be unresponsive. Otherwise, if the heartbeat delta is greater than HEARTBEAT_MIN_TIME the Server polls the client by sending a PING HEARTBEAT packet.

```

1 def heartbeat(self) -> None:
2     while self.isRunning.is_set():
3         time.sleep(HEARTBEAT_MIN_TIME)
4         with self.clientsLock:
5             clients = [k for k in self.clients.keys()]
6             for clientAddr in clients:
7                 heartbeat = self.getHeartbeat(clientAddr)
8                 delta = (datetime.now() - heartbeat).seconds
9                 if delta > HEARTBEAT_MAX_TIME:
10                    self.removeClient(
11                        clientAddr,
12                        debugStr=f"due to heartbeat timeout (last contact was
13                            {heartbeat})",
14                    )
15                elif delta > HEARTBEAT_MIN_TIME:
16                    self.queueHeartbeat(clientAddr, heartbeat=False)

```

```

1 def removeClient(self, clientAddr: tuple[str, int], debugStr="") -> None:
2     if self.checkClientExists(clientAddr):
3         cId = self.getClientId(clientAddr)
4         with self.clientsLock:
5             self.clients[clientAddr].isRunning.clear()
6             del self.clients[clientAddr]
7             if self.onClientLeave:
8                 self.onClientLeave(clientAddr, cId)

```

When a Node receives a PING HEARTBEAT packet it responds with a PONG heartbeat.

```

1 def receiveHeartbeat(
2     self, p: packet.HeartbeatPacket, addr: tuple[str, int]
3 ) -> tuple[packet.Packet, tuple[str, int]]:
4     if not p.heartbeat:
5         self.queueHeartbeat(addr, heartbeat=True)
6         pass
7     return (p, addr)

```

5.4.3.3 Callbacks The `Node` class can be initiated with an `onReceiveData` callback (taking an `addr` and some data). This callback is executed whenever a default packet is received, allowing for yielding to an *application* layer.

The `Client` class can additionally be initiated with an `onConnect` callback (taking an `addr`). The callback is called after a successful handshake is completed, allowing for a game client to begin its `mainloop`.

The `Server` class can additionally be initiated with an `onClientJoin` and `onClientLeave` callback (taking an `addr` and a `ID`). These callbacks are called whenever a client is added (i.e. completes a handshake successfully) or removed from the `Server`'s record, allowing for a game server to track its members.

5.4.3.4 ERROR Packets 5.4.3.4.1 Exceptions

The python file `udp.error` includes custom `Exceptions` for all errors defined in the Packet Specification as well as `Enum` definitions for the `Major`, and each `Minor`, error code. A base `PaperClipError` class is defined, inheriting `Exception`. Additionally, a base `Minor` enum class is defined to be used as a parent class to the various minors.

```

1 class Major(Enum):
2     ERROR = 0
3     CONNECTION = 1
4     DISCONNECT = 2
5     PACKET = 3
6
7 class Minor(Enum): pass
8
9 class PaperClipError(Exception): """Unknown error"""

```

The three `Major` error types then inherit from `PaperClipError`. The relevant `Minor` error code and their `Exceptions` are defined using the `Minor` enum and the `Minor`'s parent `Major` `Exception` respectively. The method `getConnectionError` takes a `ConnectionErrorCodes` and returns the relevant `ConnectionError`. The method `getConnectionCode` performs the reverse. This pattern is defined for all `Major` and `Minor` Codes and their relevant `Exceptions`.

```

1 # connection
2 class ConnectionErrorCodes(Minor):
3     CONNECTION = 0
4     NO_SPACE = 1
5     CERTIFICATE_INVALID = 2
6     FINISH_INVALID = 3
7
8 class ConnectionError(PaperClipError): """Handshake connection could not be
    finished"""
9
10 class NoSpaceError(ConnectionError): """Server has insufficient space to
    accept new clients"""
11
12 class CertificateInvalidError(ConnectionError): """Certificate is invalid /
    can not be validated"""
13
14 class FinishInvalidError(ConnectionError): """Finish is invalid"""
15
16 _connectionErrors = {
17     ConnectionErrorCodes.CONNECTION: ConnectionError,
18     ConnectionErrorCodes.NO_SPACE: NoSpaceError,
19     ConnectionErrorCodes.CERTIFICATE_INVALID: CertificateInvalidError,
20     ConnectionErrorCodes.FINISH_INVALID: FinishInvalidError,
21 }
22
23 def getConnectionError(minor: ConnectionErrorCodes | int) -> ConnectionError:
24     try:
25         minor = minor if isinstance(minor, Minor) else
26             ConnectionErrorCodes(minor)
27         if minor in _connectionErrors:
28             return _connectionErrors[minor]
29         else:
30             return PaperClipError
31     except ValueError:
32         return PaperClipError
33
34 def getConnectionCode(error: ConnectionError) -> ConnectionErrorCodes:
35     try:
36         return list(_connectionErrors.keys())[
37             list(_connectionErrors.values()).index(error)
38         ]
39     except ValueError:
40         return PaperClipError

```

Convenience methods allow for conversion between Enums and PaperClipErrors. The

getError method takes, either Enum or integer, Major and Minor codes and returns the relevant Exception.

```
1 def getError(major: Major | int, minor: Minor | int = 0) -> PaperClipError:
2     try:
3         major = major if isinstance(major, Major) else Major(major)
4         match major:
5             case Major.CONNECTION:
6                 return getConnectionError(minor)
7             case Major.DISCONNECT:
8                 return getDisconnectError(minor)
9             case Major.PACKET:
10                return getPacketError(minor)
11            case _:
12                return PaperClipError
13    except TypeError:
14        return PaperClipError
```

The getMinor method takes a Major and an int value minor and returns the respective Minor.

```
1 def getMinor(major: Major, minor: int) -> Minor:
2     match major:
3         case Major.CONNECTION:
4             return ConnectionErrorCodes(minor)
5         case Major.DISCONNECT:
6             return DisconnectErrorCodes(minor)
7         case Major.PACKET:
8             return PacketErrorCodes(minor)
9         case _:
10            return Minor
```

The getErrorCode method performs the reverse of the getError method, taking an PaperClipError and returning the relevant Major and Minor Enum.

```
1 def getErrorCode(error: PaperClipError) -> tuple[Major, Minor]:
2     match error:
3         case c if isinstance(c, ConnectionError):
4             return (Major.CONNECTION, getConnectionCode(error))
5         case d if isinstance(d, DisconnectError):
6             return (Major.DISCONNECT, getDisconnectCode(error))
7         case p if isinstance(p, PacketError):
8             return (Major.PACKET, getPacketCode(error))
9         case _:
10            return (Major.ERROR, Minor)
```

5.4.3.4.2 Sending an ERROR Packet

The Node's `queueError` method takes the additional `Major` and `Minor` fields. The method includes the check Node's `sequenceId` is `None` in which case it uses the value of 0 instead.

```

1 def queueError(
2     self, addr: tuple[str, int], major: error.Major | int, minor: error.Minor
      | int, flags: list[int] = [0 for _ in range(packet.FLAGS_SIZE)], data:
      bytes | None = None
3 ) -> None:
4     sId = self.getSequenceId(addr)
5     p = packet.ErrorPacket(
6         sequence_id=sId if sId is not None else 0,
7         flags=flags,
8         major=major,
9         minor=minor,
10        data=data,
11    )
12    if sId is not None:
13        self.incrementSequenceId(addr)
14    self.queuePacket(addr, p)

```

ERROR packets are automatically queued whenever a `PaperclipError` is generated by surrounding any action that could potentially yield a `relent` error with `try/except` blocks. This includes all the unpacking of all the fields in the `Packet` with each raising the relevant `PacketError`. Additionally, `ConnectionErrors` can arise during the handshake with both the `Client` and `Server` aborting and sending the relevant ERROR packet.

```

1 def receivePacket(self,) -> tuple[packet.Packet, tuple[str, int]] |
      tuple[None, None]:
2     data, addr = self.socket.recvfrom(SOCKET_BUFFER_SIZE)
3     try:
4         p = packet.unpack(data) # unpacking can yield a PacketError
5         return p, addr
6     except error.PacketError as e:
7         major, minor = error.getErrorCod(e)
8         self.queueError(addr, major, minor)
9         return None, None

1 def receive(self, p: packet.Packet, addr: tuple[str, int]) ->
      tuple[packet.Packet, tuple[str, int]] | None:
2     if p is not None:
3         if self.handleFlags(p, addr):
4             match p.packet_type:
5                 # packet type cases omitted for clarity
6                 case _: # unknown packet type
7                     self.queueError(
8                         addr,
9                         major=error.Major.PACKET,

```

```

10         minor=error.PacketErrorCodes.PACKET_TYPE,
11         data=p.sequence_id,
12     )

```

5.4.3.4.3 Receiving an ERROR Packet

When receiving an ERROR packet, the `receive` method passes the packet to the `receiveError` method within a `try/except` block. The `receiveError` method derives and raises the relevant `PaperclipError` from the packet's Major and Minor fields. The data field is used to append additional information to the derived `Exception`. This causes the `try/except` block to pass the error to `handleError` which, in turn, passes the error to the relevant error handler.

```

1 case packet.Type.ERROR:
2     try:
3         return self.receiveError(p, addr)
4     except error.PaperClipError as e:
5         self.handleError(p, addr, e)

1 def receiveError(self, p: packet.ErrorPacket, addr: tuple[str, int]) -> None:
2     raise error.getError(p.major, p.minor)(p.data)

1 def handleError(self, p: packet.ErrorPacket, addr: tuple[str, int], e:
    error.PaperClipError) -> None:
2     match e:
3         case error.ConnectionError():
4             self.handleConnectionError(p, addr, e)
5         case error.DisconnectError():
6             self.handleDisconnectError(p, addr, e)
7         case error.PacketError():
8             self.handlePacketError(p, addr, e)
9         case _:
10            raise e

```

If a Node receives a `ConnectionError` the Node abort's the connection and calls the `quit` method to gracefully stop threads.

```

1 def handleConnectionError(self, p: packet.ErrorPacket, addr: tuple[str,
    int], e: error.ConnectionError) -> None:
2     match e:
3         case error.NoSpaceError():
4             return self.quit("no server space", e)
5         case error.CertificateInvalidError():
6             return self.quit("invalid certificate", e)
7         case error.FinishInvalidError():
8             return self.quit("invalid finish", e)
9         case _:
10            raise e

```

The `handleDisconnectError` provides a method to be overridden by the `Client` and `Server`.

```
1 def handleDisconnectError(  
2     self, p: packet.ErrorPacket, addr: tuple[str, int], e:  
3     error.DisconnectError) -> None:  
4     match e:  
5         case error.ServerDisconnectError:  
6             pass # overwrite  
7         case error.ClientDisconnectError:  
8             pass # overwrite  
9         case _:  
10            raise e
```

If a `Node` receives a `PacketError` it performs no additional actions.

```
1 def handlePacketError(self, p: packet.ErrorPacket, addr: tuple[str, int],  
2     e: error.PacketError) -> None:  
3     pass
```

5.4.3.5 Disconnects The `Node` provides an overridable convenience method for sending a `DisconnectError`. Both the `Client` and `Server` override this method to replace the minor with `error.DisconnectErrorCodes.CLIENT_DISCONNECT` and `error.DisconnectErrorCodes.SERVER_DISCONNECT` respectively.

```
1 def queueDisconnect(self, addr: tuple[str, int], flags: list[bool] = [0 for _  
2     in range(packet.FLAGS_SIZE)], data: bytes | None = None) -> None:  
3     self.queueError(  
4         addr,  
5         flags=flags,  
6         major=error.Major.DISCONNECT,  
7         minor=error.DisconnectErrorCodes.DISCONNECT,  
8         data=data,  
9     )
```

5.4.3.5.1 Client Disconnect

The `Client` overrides the `handleDisconnectError` method to call `_quit` on a `ServerDisconnectError`. The methods `quit` and `_quit` perform the same actions of gracefully stopping the threads but `quit` also includes sending a `ClientDisconnectError` to the server **before** terminating. As the `Server` has initiated the termination, `_quit` is called to skip sending the error.

```
1 def handleDisconnectError(  
2     self, p: packet.ErrorPacket, addr: tuple[str, int], e:  
3     error.DisconnectError  
4 ) -> None:
```



```

4         match e:
5             case error.ServerDisconnectError():
6                 self._quit(e)
7             case error.ClientDisconnectError():
8                 pass # should not react to client disconnect
9             case _:
10                raise e

```

5.4.3.5.2 Server Disconnect

The `Client` overrides the `handleDisconnectError` method to call `removeClient` to close the client instance `Node`. Unlike the `Client`, the `Server` does not terminate.

```

1 def handleDisconnectError(
2     self, p: packet.ErrorPacket, addr: tuple[str, int], e:
3         error.DisconnectError
4 ) -> None:
5     match e:
6         case error.ServerDisconnectError():
7             pass # should not react to server disconnect
8         case error.ClientDisconnectError():
9             self.removeClient(addr, "The client has closed")
10        case _:
11            raise e

```

The `Server` also overrides the `queueDisconnect` to send a `ServerDisconnectError` to all clients. This is called on a `Server` quit (along with the termination of threads).

```

1 def queueDisconnect(self, flags: list[bool] = [0 for _ in
2     range(packet.FLAGS_SIZE)], data: bytes | None = None):
3     with self.clientsLock:
4         clientAddrs = [addr for addr in self.clients]
5         for addr in clientAddrs:
6             self.queueError(
7                 addr,
8                 flags=flags,
9                 major=error.Major.DISCONNECT,
10                minor=error.DisconnectErrorCodes.SERVER_DISCONNECT,
11                data=data,
12            )

```

5.4.4 Iteration 4

5.4.4.1 DotEnv Variables previously defined as `CONSTs` are moved into a central `.env` file. This allows for easier value management.

```

1 # .env
2
3 # udp
4 S_HOST=127.0.0.1
5 S_PORT=2024
6 C_HOST=127.0.0.1
7 C_PORT=2025
8 ## node
9 SOCKET_BUFFER_SIZE = 1024
10 SEND_SLEEP_TIME = 0.1
11 QUEUE_TIMEOUT = 10
12 SOCKET_TIMEOUT = 20
13 ## server
14 HEARTBEAT_MAX_TIME = 120
15 HEARTBEAT_MIN_TIME = 30
16 MAX_CLIENTS
17 ## auth
18 ORG_NAME = Paperclip
19 COMMON_NAME = 127.0.0.1
20 ## utils
21 MAX_FRAGMENT_SIZE = 988
22
23 # client
24 TCP_PORT = 5000
25
26 # app
27 FLASK_APP = server
28 PRUNE_TIME = 58
29 SECRET_KEY = MyVerySecretKey
30 SQLALCHEMY_DATABASE_URI = mysql://root:root@localhost:3306/paperclip
31
32 # debug
33 DEBUG = True

```

The variables can then be loaded from the `os.environ` by first calling `dotenv.load_dotenv(".env")`. This is done each each package's `__init__` file.

The `udp.__init__` loads all the relevant variables for the `udp` package to constants, which can then in turn be imported in each script using `from . import VAR_NAME_ONE, VAR_NAME_TWO, VAR_NAME_N` (where `VAR_NAME` is the name of the `CONST` to be imported)

```

1 import os
2 import dotenv
3
4 dotenv.load_dotenv(".env")
5 S_HOST = os.environ.get("S_HOST")

```

```

6 S_PORT = int(os.environ.get("S_PORT"))
7 C_HOST = os.environ.get("C_HOST")
8 C_PORT = int(os.environ.get("C_PORT"))
9 # node
10 SOCKET_BUFFER_SIZE = int(os.environ.get("SOCKET_BUFFER_SIZE"))
11 SEND_SLEEP_TIME = float(os.environ.get("SEND_SLEEP_TIME"))
12 QUEUE_TIMEOUT = int(os.environ.get("QUEUE_TIMEOUT"))
13 SOCKET_TIMEOUT = int(os.environ.get("SOCKET_TIMEOUT"))
14 # server
15 HEARTBEAT_MAX_TIME = int(os.environ.get("HEARTBEAT_MAX_TIME"))
16 HEARTBEAT_MIN_TIME = int(os.environ.get("HEARTBEAT_MIN_TIME"))
17 MAX_CLIENTS = (
18     int(os.environ.get("MAX_CLIENTS"))
19     if os.environ.get("MAX_CLIENTS") is not None
20     else float("inf")
21 )
22 # auth
23 ORG_NAME = os.environ.get("ORG_NAME")
24 COMMON_NAME = os.environ.get("COMMON_NAME")
25 # utils
26 MAX_FRAGMENT_SIZE = int(os.environ.get("MAX_FRAGMENT_SIZE"))

```

5.4.4.2 Logging A `logging.Logger` is used to provide runtime logging of system outputs. The logging module provides the option of logging with different levels (e.g. `DEBUG`, `INFO`, `ERROR`) allowing different situations to provide different outputs. A logger is initiated in the `udp.__init__` with a default log level of `DEBUG`. Additionally, `bcolors` includes a various ASCII color codes to allow for rich-color output to the console.

```

1 import logging
2 import sys
3
4 class bcolors:
5     HEADER = "\033[95m"
6     OKBLUE = "\033[94m"
7     OKCYAN = "\033[96m"
8     OKGREEN = "\033[92m"
9     WARNING = "\033[93m"
10    FAIL = "\033[91m"
11    ENDC = "\033[0m"
12    BOLD = "\033[1m"
13    UNDERLINE = "\033[4m"
14
15 logger = logging.getLogger(__name__)
16 logger.setLevel(logging.DEBUG)

```

A `StreamHandler` `printHandler` is defined to output to `sys.stdout` with the default level of `INFO` allowing all messages with `INFO` or higher (i.e. not `DEBUG`) to be printed to the console. `printHandler` is given a `logging.Formatter` so that the `threadName` (colored blue) is recorded with the inputted message.

```
1 printHandler = logging.StreamHandler(sys.stdout)
2 printHandler.setLevel(logging.INFO)
3 printHandler.setFormatter(
4     logging.Formatter("{bcolors.OKBLUE}%(threadName)s{bcolors.ENDC} -
5         %(message)s")
6 )
7 logger.addHandler(printHandler)
```

A `FileHandler` `fileHandler` is defined to output to `paperclip.log` with the default level `DEBUG` meaning all messages are recorded. `fileHandler` is given a `Formatter` such that each message contains the `asctime`, `levelname` and `threadName` in addition to the inputted message. Additionally, a custom `logging.Filter` `ColorFilter` is defined to remove any ASCII color codes from messages allowing for log messages to include color codes for **only** the console output.

```
1 class ColorFilter(logging.Filter):
2     colorCodes = [
3         getattr(bcolors, attr) for attr in dir(bcolors) if not
4             attr.startswith("__")
5     ]
6
7     def filter(self, record: logging.LogRecord) -> bool:
8         for color in self.colorCodes:
9             record.msg = record.msg.replace(color, "")
10         return True
11
12 fileHandler = logging.FileHandler("paperclip.log")
13 fileHandler.setLevel(logging.DEBUG)
14 fileHandler.addFilter(ColorFilter())
15 fileHandler.setFormatter(
16     logging.Formatter("%(asctime)s - %(levelname)s - %(threadName)s -
17         %(message)s")
18 )
19 logger.addHandler(fileHandler)
```

The `logger.info` method is used to record typical behaviors.

```
1 logger.info(f"{bcolors.OKBLUE}> {addr} :{bcolors.ENDC}
2     {bcolors.OKCYAN}{p}{bcolors.ENDC}") # INFO: log outgoing packet
3 logger.info(f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC}
4     {bcolors.OKCYAN}{p}{bcolors.ENDC}") # INFO: log incoming packet
```

The `logger.warning` method is used to record whenever something has not occurred as expected (without causing an error)

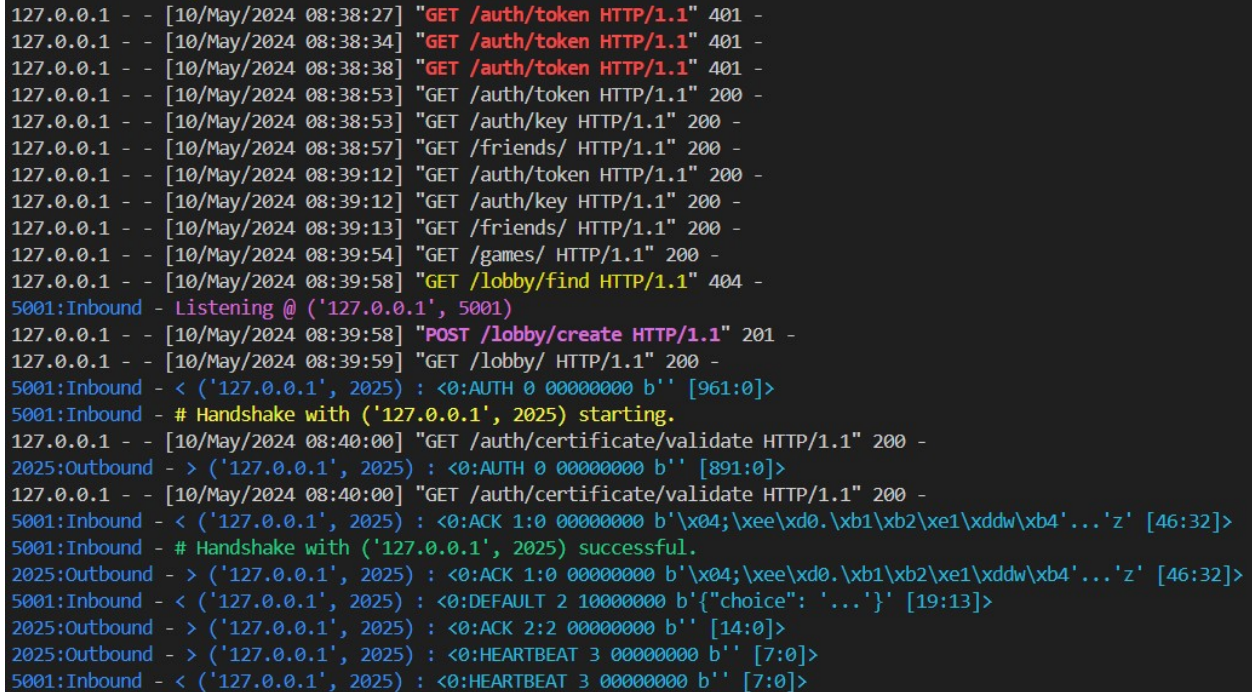
```
1 logger.warning(f"\tInvalid checksum: {p}") # WARNING: log invalid checksum
```

The `logger.error` method is used to record whenever an error occurs.

```
1 logger.error(f"{bcolors.FAIL}# > {bcolors.ENDC}{bcolors.OKBLUE}{addr}
    :{bcolors.ENDC} {bcolors.FAIL}{type(e).__name__}:{e.args[0] if len(e.args)
    > 0 else ''}{p}{bcolors.ENDC}") # WARNING: log a PaperclipError
```

The `logger.critical` method is used to record whenever a **critical** error occurs meaning the program is unable to continue running.

```
1 logger.critical(f"Invalid peer cert {p.certificate}") # CRITICAL: log invalid
    (server) certificate yielding an abort
```



```
127.0.0.1 - - [10/May/2024 08:38:27] "GET /auth/token HTTP/1.1" 401 -
127.0.0.1 - - [10/May/2024 08:38:34] "GET /auth/token HTTP/1.1" 401 -
127.0.0.1 - - [10/May/2024 08:38:38] "GET /auth/token HTTP/1.1" 401 -
127.0.0.1 - - [10/May/2024 08:38:53] "GET /auth/token HTTP/1.1" 200 -
127.0.0.1 - - [10/May/2024 08:38:53] "GET /auth/key HTTP/1.1" 200 -
127.0.0.1 - - [10/May/2024 08:38:57] "GET /friends/ HTTP/1.1" 200 -
127.0.0.1 - - [10/May/2024 08:39:12] "GET /auth/token HTTP/1.1" 200 -
127.0.0.1 - - [10/May/2024 08:39:12] "GET /auth/key HTTP/1.1" 200 -
127.0.0.1 - - [10/May/2024 08:39:13] "GET /friends/ HTTP/1.1" 200 -
127.0.0.1 - - [10/May/2024 08:39:54] "GET /games/ HTTP/1.1" 200 -
127.0.0.1 - - [10/May/2024 08:39:58] "GET /lobby/find HTTP/1.1" 404 -
5001:Inbound - Listening @ ('127.0.0.1', 5001)
127.0.0.1 - - [10/May/2024 08:39:58] "POST /lobby/create HTTP/1.1" 201 -
127.0.0.1 - - [10/May/2024 08:39:59] "GET /lobby/ HTTP/1.1" 200 -
5001:Inbound - < ('127.0.0.1', 2025) : <0:AUTH 0 00000000 b'' [961:0]>
5001:Inbound - # Handshake with ('127.0.0.1', 2025) starting.
127.0.0.1 - - [10/May/2024 08:40:00] "GET /auth/certificate/validate HTTP/1.1" 200 -
2025:Outbound - > ('127.0.0.1', 2025) : <0:AUTH 0 00000000 b'' [891:0]>
127.0.0.1 - - [10/May/2024 08:40:00] "GET /auth/certificate/validate HTTP/1.1" 200 -
5001:Inbound - < ('127.0.0.1', 2025) : <0:ACK 1:0 00000000 b'\x04;\xee\xd0.\xb1\xb2\xe1\xddw\xb4'... 'z' [46:32]>
5001:Inbound - # Handshake with ('127.0.0.1', 2025) successful.
2025:Outbound - > ('127.0.0.1', 2025) : <0:ACK 1:0 00000000 b'\x04;\xee\xd0.\xb1\xb2\xe1\xddw\xb4'... 'z' [46:32]>
5001:Inbound - < ('127.0.0.1', 2025) : <0:DEFAULT 2 10000000 b'{"choice": "..."}' [19:13]>
2025:Outbound - > ('127.0.0.1', 2025) : <0:ACK 2:2 00000000 b'' [14:0]>
2025:Outbound - > ('127.0.0.1', 2025) : <0:HEARTBEAT 3 00000000 b'' [7:0]>
5001:Inbound - < ('127.0.0.1', 2025) : <0:HEARTBEAT 3 00000000 b'' [7:0]>
```

Figure 3: example of server console output

5.4.4.3 Database Models The database models are implemented as SQLAlchemy `db.models`.

```
1 uri = os.environ.get("SQLALCHEMY_DATABASE_URI") # get uri from .env
2 _init = False
3 if not database_exists(uri): # create database if not exists
4     _init = True
```

```

5     create_database(uri)
6 app.config["SQLALCHEMY_DATABASE_URI"] = uri
7
8 db.init_app(app)
9
10 with app.app_context():
11     db.create_all() # create all tables
12
13 if _init: # if database was created
14     with app.app_context(): # create some dummy data
15         # init games
16         from rps import ID, NAME, MIN_PLAYERS, MAX_PLAYERS
17         Statement.createGame(ID, NAME, MIN_PLAYERS, MAX_PLAYERS)
18         # example accounts
19         m = Statement.createAccount("Mario", "ItsAMe123")
20         p = Statement.createAccount("Peach", "MammaMia!")
21         b = Statement.createAccount("Bowser", "M4r10SucK5")
22         Statement.createFriends(m.id, p.id)
23         Statement.createFriends(p.id, b.id)

```

The models were largely implemented according to the ERD with some additional fields. The `Statement` class contains various convenience methods for acting on the database (i.e. getting, creating and deleting rows).

5.4.4.3.1 Friends Model

The `Friends` class is implemented according to the ERD.

```

1 class Friends(db.Model):
2     account_one_id = db.Column(
3         db.Integer, db.ForeignKey("account.id"), primary_key=True
4     )
5     account_two_id = db.Column(
6         db.Integer, db.ForeignKey("account.id"), primary_key=True
7     )

```

The `Statement` for creating friends ensures that `idOne < idTwo`. This allows for easier look-ups of the data as the order of the given accountIds can be derived.

```

1 class Statement:
2     @staticmethod
3     def getFriends(accountId: int) -> list[Account]: # retrieve list of
4         Accounts who are Friends with id
5         friends = Friends.query.filter( # filter where either account_one_id
6             or account_two_id is accountId
7             (Friends.account_one_id == accountId)
8             | (Friends.account_two_id == accountId)
9         )

```

```

7         )
8         friends = [ # get the account_id of the other account
9                     friend.account_one_id
10                    if friend.account_one_id != accountId
11                    else friend.account_two_id
12                    for friend in friends
13                ]
14        friends = [Statement.getAccount(id) for id in friends] # get list of
15        accounts
16        return friends
17
18    @staticmethod
19    def createFriends(accountIdOne: int, accountIdTwo: int) -> Friends: #
20        create, commit and return Friends
21        # ensure that idOne < idTwo for index efficiency & easier look-up
22        idOne = min(accountIdOne, accountIdTwo)
23        idTwo = max(accountIdOne, accountIdTwo)
24        friends = Friends(account_one_id=idOne, account_two_id=idTwo)
25        db.session.add(friends)
26        db.session.commit()
27        return friends
28
29    @staticmethod
30    def removeFriends(accountIdOne: int, accountIdTwo: int) -> bool: # delete
31        Friends. True on success.
32        # ensure that idOne < idTwo
33        idOne = min(accountIdOne, accountIdTwo)
34        idTwo = max(accountIdOne, accountIdTwo)
35        friends = Friends.query.filter(
36            (Friends.account_one_id == idOne) & (Friends.account_two_id ==
37            idTwo)
38        )
39        if friends is not None:
40            friends.delete() # delete
41            db.session.commit()
42            return True
43        else:
44            return False

```

5.4.4.3.2 Game Model

The **Game** model was expanded to also include a string **Name**, for better usability, as well as integer **min_players** and **max_players** fields so a game server is able to start the game after enough members have joined as well as prevent too many players from joining respectively. The **max_players** is also used so the API Server (via the LobbyHandler) can tell which

Lobbies are full.

```
1 class Game(db.Model):
2     id = db.Column(db.Integer, primary_key=True)
3     name = db.Column(db.String(255), unique=True, nullable=False)
4     min_players = db.Column(db.Integer, default=1)
5     max_players = db.Column(db.Integer)
```

Statements are defined to allow for the creation and retrieval of Games. The `getGames` method allows for all games to be retrieved.

```
1 class Statement:
2     @staticmethod
3     def getGame(gameId: int) -> Game:
4         return Game.query.filter_by(id=gameId).scalar() # retrieve Game by id
5
6     @staticmethod
7     def getGames() -> list[Game]:
8         return Game.query.all() # retrieve all Game
9
10    @staticmethod
11    def createGame(id:int, name:str, min_players:int, max_players:int) ->
12        Game: # create, commit and return Game
13        game = Game(id=id, name=name, min_players=min_players,
14                    max_players=max_players)
15        db.session.add(game)
16        db.session.commit()
17        return game
18
19    @staticmethod
20    def findGame(gameName: str) -> Game | None:
21        return Game.query.filter_by(name=gameName).scalar() # retrieve Game
22            by name
```

5.4.4.3.3 Account Model

The Account model was expanded to also include `private_key` and `public_key` which are DER bytes formatted versions of each account's RSA key. SQLAlchemy also allows for models to contain additional methods for use with instance variables. This allowed for security features such as the hashing of passwords and generation of RSA key to be performed on a new instance before it is committed to the database.

```
1 class Account(db.Model):
2     id = db.Column(db.Integer, primary_key=True)
3     username = db.Column(db.String(255), unique=True, nullable=False)
4     password = db.Column(db.String(162), nullable=False)
5     private_key = db.Column(db.LargeBinary(1337)) # DER bytes private RSA key
6     public_key = db.Column(db.LargeBinary(294)) # DER bytes public RSA key
```



```

7
8     def hashPassword(self, password: str) -> None:
9         self.password = generate_password_hash(password)
10
11     def verifyPassword(self, password: str) -> bool:
12         return check_password_hash(self.password, password)
13
14     def generateKey(self, password: bytes) -> None:
15         k = auth.generateRsaKey()
16         self.private_key = auth.getDerFromRsaPrivate(k, password) # encrypts
17             DER with password for security
18         self.public_key = auth.getDerFromRsaPublic(k.public_key())
19
20     @staticmethod
21     def decryptKey(self, key: bytes, password: bytes) ->
22         auth.rsa.RSAPublicKey:
23         k = auth.getRsaPrivateFromDer(key, password)
24         return k

```

Statements are defined to allow for the creation and retrieval of Accounts. The `createAccount` method ensures that the password is hashed as well as generating a RSA key for the Account before the it is committed.

```

1 class Statement:
2     # get
3     @staticmethod
4     def getAccount(userId: int) -> Account:
5         return Account.query.filter_by(id=userId).scalar() # retrieve Account
6             by id
7
8     # create
9     @staticmethod
10    def createAccount(username: str, password: str) -> Account: # create,
11        commit and return Account
12        account = Account(username=username)
13        account.hashPassword(password) # hash password
14        account.generateKey(password.encode()) # generate RSA key
15        db.session.add(account)
16        db.session.commit()
17        return account
18
19    # find
20    @staticmethod
21    def findAccount(username: str) -> Account | None:
22        return Account.query.filter_by(username=username).scalar() # retrieve
23            Account by username

```

5.4.4.3.4 Lobby Model

Finally, upon reflection, the `Lobby` and `LobbyMembers` models and their behaviour were better suited as python class instances (i.e. were removed from the database). The `Lobby` model was refactored into a `Lobby` class which is responsible for initiating and running a game server instance. In addition to this, the `Lobby` class is responsible for tracking and reporting lobby members, so no `LobbyMembers` class is needed.

```
1 def isNotFull(self) -> bool:
2     return self.gameServer.isNotFull()
3
4 def isEmpty(self) -> bool:
5     return len(self.members) == 0
```

A `LobbyHandler` class was created to manage the creation of Lobbys and the `API Server` uses this when dispatching new Lobbys rather than creating them directly. The `LobbyHandler` is also responsible for *pruning* lobbies. In a `pruneThread` the `LobbyHandler` iterates over all of the `Lobby` instances and checks their heartbeat (in a similar fashion to how a `udp.Server` removes old clients). The `prune` method creates a copy of `lobbies` list to iterate over (rather than iterating over the `lobbies` themselves). It can be assumed that any Lobbies created during the execution of the `prune` loop will not be old enough to be pruned. If a `Lobby` has contained no members for some `PRUNE_TIME` (60 seconds) the `LobbyHandler` stops and removes it to free up resources.

```
1 def prune(self) -> None:
2     while self.isRunning:
3         with self.lobbiesLock:
4             lobbies = self.lobbies.copy() # create copy to iterate over
                                           # for better thread-safety.
5         for lobby in lobbies:
6             if lobby.isPrune():
7                 logger.info(
8                     f"{bcolors.FAIL}# Lobby {lobby} was removed due to
                        PRUNE
                        (delta={lobby._heartbeatDelta()}){bcolors.ENDC}"
9                 )
10                self.deleteLobby(lobby.addr)
11                time.sleep(PRUNE_TIME)
```

The `Lobby` contains the `isPrune` method allowing the `LobbyHandler` to determine if the `Lobby` should be deleted.

```
1 def isPrune(self) -> bool:
2     if isinstance(self.heartbeat, datetime.datetime):
3         delta = self._heartbeatDelta()
4         if delta > PRUNE_TIME: # check if server has been empty for >
                                PRUNE_TIME
5             return True
```

```

6     return False
7
8 def _heartbeatDelta(self) -> int:
9     return (datetime.datetime.now() - self.heartbeat).seconds

```

When a client joins the Lobby it sets the heartbeat to `true` to indicate it has active members. When a client leaves the Lobby, if the Lobby's has no members, it sets the heartbeat to `now()`.

```

1 def onJoin(self, addr: tuple[str, int], accountId: int) -> None:
2     self.members.append(accountId)
3     self.heartbeat = True
4
5 def onLeave(self, addr: tuple[str, int], accountId: int) -> None:
6     self.members.remove(accountId)
7     if self.isEmpty():
8         self.heartbeat = datetime.datetime.now()

```

5.4.4.4 RESTful Server The RESTful Server was implemented using as a Flask app.

5.4.4.4.1 API Authentication

HTTPBasicAuth allows for easy authentication with a username and password and can restrict access to certain endpoints unless authentication is provided (using the `@auth.login_required` decorator). JSON Web Tokens (JWT) are used for session tokens, allowing a user to instead request and use a token for the rest of the session (or until the token expires) instead of using a username and password. This can help mitigate against any man-in-the-middle attacks as, if a token is successfully intercepted, it will only be useable for a limited time and the accounts credentials are not exposed.

The `verifyPassword` method used by `auth` first checks if it has been given a token. Otherwise, the method attempts to validate with the username and password.

```

1 auth = HTTPBasicAuth()
2
3 @auth.verify_password
4 def verifyPassword(username: str, password: str) -> bool:
5     account = Statement.validateToken(username) # check token
6     if not account: # if token not valid
7         account = Statement.findAccount(username=username) # check account
8         if not account or not account.verifyPassword(
9             password
10        ): # if account not exist or wrong password
11             return False
12     g.account = account # store (until overwrite) in flask globals
13     return True

```

JWT tokens are generated using the `Account` class's `generateToken` method. The tokens include the `Account.id` and remain valid for `expiration` seconds (default to 600).

```
1 def generateToken(self, expiration: int = 600) -> str:
2     data = {
3         "id": self.id,
4         "exp": datetime.datetime.now() +
5             datetime.timedelta(seconds=expiration),
6     }
7     token = jwt.encode(data, current_app.config["SECRET_KEY"],
8         algorithm="HS256")
9     return token
```

Tokens are validated using `Statement.validateToken` which calls the `Account.validateToken` static method.

```
1 class Statement:
2     @staticmethod
3     def validateToken(token: str) -> Account | None:
4         return Account.validateToken(token)
```

The `Account.validateToken` method performs the JWT decode function on the token. This includes checks for token expiry. On a success it returns the `Account` with the relevant id.

```
1 @staticmethod
2 def validateToken(token: str):
3     try:
4         data = jwt.decode(
5             token,
6             current_app.config["SECRET_KEY"],
7             leeway=datetime.timedelta(seconds=10),
8             algorithms=["HS256"],
9         )
10    except:
11        return None
12    account = Statement.getAccount(data.get("id"))
13    return account
```

5.4.4.4.2 Endpoints

The endpoints are implemented according to the API specification.

5.4.4.4.2.1 Auth

The `createAccount` method is exposed at `/auth/register` and accepts **only** POST requests. The method takes a username and password field from the request's JSON and creates a new account. The `Account.id` and `Account.username` are returned with the HTTP code 201 to indicate successful account creation.

```

1 @main.route("/auth/register", methods=["POST"])
2 def createAccount():
3     username = request.json.get("username")
4     password = request.json.get("password")
5     if not (username or password): # check not null
6         abort(400) # missing args
7     if Statement.findAccount(username): # check if account exists
8         abort(400) # account already exists
9     account = Statement.createAccount(username, password)
10    return jsonify({"account-id": account.id, "username": account.username}),
    201

```

The `getAuthToken` method is exposed at `/auth/token` and accepts **only** GET requests. The method generates and returns a token derived from the logged-in `Account`.

```

1 @main.route("/auth/token")
2 @auth.login_required
3 def getAuthToken():
4     return jsonify({"token": g.account.generateToken()})

```

The `getKey` method is exposed at `/auth/key` and accepts **only** GET requests. The method retrieves the DER `private_key` associated with the logged-in `Account`. The key is `base64` encoded as a sanitation step to ensure it can be encoded in URL safe JSON.

```

1 @main.route("/auth/key")
2 @auth.login_required
3 def getKey():
4     return jsonify(
5         {
6             "key": base64.encodebytes(g.account.private_key).decode(),
7             "account-id": g.account.id,
8         }
9     )

```

5.4.4.4.2 Friends

The `getFriends` method is exposed at `/friends` and accepts **only** GET requests. The method returns a list of dictionaries of all `Account.id` and `Account.username` where the `Account` is friends with the logged-in account.

```

1 @main.route("/friends/")
2 @auth.login_required
3 def getFriends():
4     friends = Statement.getFriends(g.account.id)
5     return jsonify(
6         {
7             "friends": [

```

```

8         {"id": account.id, "username": account.username} for account
          in friends
9     ]
10 }
11 )

```

The `addFriend` method is exposed at `/friends/add` and accepts **only** POST request. The method derives two accounts, one from the logged-in account and the other from the username field in the request's JSON. The method then creates a new `Friends` entry and returns the `Account.id` and `Account.username` of both `Accounts` along with the HTTP code 201.

```

1 @main.route("/friends/add", methods=["POST"])
2 @auth.login_required
3 def addFriend():
4     username = request.json.get("username")
5     if username is None:
6         abort(400) # missing args
7     account = g.account
8     other = Statement.findAccount(username)
9     if other is None:
10         abort(404)
11     Statement.createFriends(account.id, other.id)
12     return jsonify(
13         {
14             "account": {"id": account.id, "username": account.username},
15             "other": {"id": other.id, "username": other.username},
16         }
17     ), 201

```

The `removeFriend` method is exposed at `/friend/remove` and accepts **only** DELETE requests. The method derives two accounts, in the same way as `addFriend` and deletes the `Friends` from the database. The method returns 204 to indicate a successful deletion

```

1 @main.route("/friend/remove", methods=["DELETE"])
2 @auth.login_required
3 def removeFriend():
4     username = request.json.get("username")
5     if username is None:
6         abort(400) # missing args
7     account = g.account
8     other = Statement.findAccount(username)
9     if other is None:
10         abort(404) # no such account
11     success = Statement.removeFriends(account.id, other.id)
12     if success:
13         return jsonify(data=[]), 204

```

```

14     else:
15         abort(404) # no such friends

```

5.4.4.4.2.3 Games

The `getGames` method is exposed at `/games` and accepts **only** GET request. The method returns a list of all available games.

```

1 @main.route("/games/")
2 @auth.login_required
3 def getGames():
4     return jsonify({game.id: game.name for game in Statement.getGames()})

```

5.4.4.4.2.4 Lobby

The `getLobby` method is exposed at `/lobby` and accepts **only** GET requests. The method derives a Lobby from the `LobbyHandler`, using the the `lobby-id` in the request's JSON, and returns the Lobby's `id`, `addr` and `gameId`.

```

1 @main.route("/lobby/")
2 @auth.login_required
3 def getLobby():
4     lobbyId = request.json.get("lobby-id")
5     if not lobbyId:
6         abort(400) # missing args
7     lobby = lobbyHandler.getLobby(lobbyId)
8     return jsonify(
9         {"lobby-id": lobby.id, "lobby-addr": lobby.getAddr(), "game-id":
10         lobby.gameId}
11 )

```

The `getLobbies` method at `/lobby/all` compiles all lobbies currently in the `LobbyHandler`. It returns a variety of information on each lobby in a list of dictionaries. The information includes the `lobby-id`, the game (with `game-id` and `game-name`) associated with the Lobby, the `max size` and if the lobby `is-full`.

```

1 @main.route("/lobby/all")
2 @auth.login_required
3 def getLobbies():
4     lobbies = LobbyHandler.getAll()
5     games = {game.id: game.name for game in Statement.getGames()}
6     data = lambda lobby: {
7         "game": {"game-id": lobby.game_id, "game-name": games[lobby.game_id]},
8         "size": Statement.getLobbySize(lobby.id),
9         "is-full": Statement.getIsLobbyFree(lobby.id),
10    }
11    return jsonify({lobby.id: data(lobby) for lobby in lobbies})

```

The `findLobby` method at `/lobby/find` finds a `Lobby` instance with available space (i.e. `isNotFull`) using either the `gameId` or `gameName` provided in the request's JSON. The method returns the `lobby-id`, `lobby-addr` and `game-id` in a dictionary.

```
1 @main.route("/lobby/find")
2 @auth.login_required
3 def findLobby():
4     gameId = request.json.get("game-id")
5     gameName = request.json.get("game-name")
6     if not (gameId or gameName): # check args
7         abort(400) # missing args
8     game = None
9     if gameId: # check gameId not null
10         game = Statement.getGame(gameId)
11     if not game: # check gameId null
12         if gameName: # check gameName not null
13             game = Statement.findGame(gameName)
14     if not game: # check game null
15         abort(404) # no game found
16     lobby = lobbyHandler.findLobbies(game.id)
17     lobby = lobby[0] if len(lobby) > 0 else None
18     if lobby is not None:
19         return jsonify(
20             {
21                 "lobby-id": lobby.id,
22                 "lobby-addr": lobby.getAddr(),
23                 "game-id": lobby.gameId,
24             }
25         )
26     else:
27         abort(404)
```

The `createLobby` method is exposed at `/lobby/create` and accepts **only** POST requests. The method creates a new `Lobby` instance using the `LobbyHandler` and the `game.id` derived from either the `game-id` or `game-name` included in the request's JSON.

```
1 @main.route("/lobby/create", methods=["POST"])
2 @auth.login_required
3 def createLobby():
4     gameId = request.json.get("game-id")
5     gameName = request.json.get("game-name")
6     if not (gameId or gameName): # check args
7         abort(400) # missing args
8     game = None
9     if gameId: # check gameId not null
10         game = Statement.getGame(gameId)
```



```

11     if not game: # check gameId null
12         if gameName: # check gameName not null
13             game = Statement.findGame(gameName)
14     if not game: # check game null
15         abort(404) # no game found
16     addr = _getAddr()
17     lobby = lobbyHandler.createLobby(addr, game.id)
18     return jsonify(
19         {"lobby-id": lobby.id, "lobby-addr": lobby.getAddr(), "game-id":
20         lobby.gameId}
21     ), 201

```

The `getMembers` method at `/lobby/members/` returns a dictionary of all members of all lobbies.

```

1 @main.route("/lobby/members")
2 @auth.login_required
3 def getMembers():
4     return jsonify(lobbyHandler.getMembers())
5
6 # LobbyHandler.getMembers
7 def getMembers(self) -> dict[int, list[int]]:
8     with self.lobbiesLock:
9         return {lobby.id: lobby.members for lobby in self.lobbies}

```

The `getFriendLobbies` method at `/lobby/friends` returns all Lobbies which contain an Account which is Friends with the logged-in Account as long as the Lobby has space. It calls `lobbyHandler.getMember` to retrieve the relevant Lobbies.

```

1 @main.route("/lobby/friends")
2 @auth.login_required
3 def getFriendLobbies():
4     friends = Statement.getFriends(g.account.id)
5     lobbyInfo = lambda lobby: {
6         "lobby-id": lobby.id,
7         "game-id": lobby.gameId,
8         "game-name": Statement.getGame(lobby.gameId).name,
9     }
10    accountInfo = lambda account: {
11        "account-id": account.id,
12        "username": account.username,
13    }
14    lobbies = [
15        {
16            "account": accountInfo(account),
17            "lobbies": [

```

```

18         lobbyInfo(lobby) for lobby in
19             lobbyHandler.getMember(account.id)
20     ],
21     }
22     for account in friends
23         if len(lobbyHandler.getMember(account.id)) > 0
24 ]
25 return jsonify(lobbies)

```

The `LobbyHandler.getMember` method returns a list of all Lobbys an containing an `Account` with `Account.id == accountId`. It performs an additional check to only return Lobbys with available space (i.e. `isNotFull()`).

```

1 def getMember(self, accountId: int) -> list[Lobby]:
2     with self.lobbiesLock:
3         return [
4             lobby
5             for lobby in self.lobbies
6             if lobby.isNotFull() and accountId in lobby.getMembers()
7         ]

```

5.4.4.5 Certificates and Handshake Update The `validateCert` method is exposed at `auth/certificate/validate` and accepts **only** GET requests. The method a DER certificate from the request's JSON and (after `base64` decoding) converted to a `x509.Certificate` instance. The `Account` can then be derived using the `account-id` from the certificate attributes and the associated DER `Account.public_key` can be retried and converted to an `rsa.RSAPublicKey` instance. If an `account-id` is not present the certificate is checked against the Server's RSA key. The validity can then be checked and returned along with the derived `Account.id`.

```

1 @main.route("/auth/certificate/validate")
2 def validateCert():
3     valid = False
4     certificate = request.json.get("certificate")
5     certificate = base64.decodebytes(certificate.encode()) # base64 decode
6     if certificate is not None:
7         certificate = udp.auth.getCertificateFromDer(certificate) # get
8             x509.Certificate instance
9         attributes = udp.auth.getUserCertificateAttributes(certificate)
10        if attributes["account-id"] is not None:
11            account = Statement.getAccount(attributes["account-id"]) # get
12                Account instance
13            publicKey = udp.auth.getRsaPublicFromDer(account.public_key) #
14                get rsa.RSAPublicKey instance

```

```

12     else:
13         publicKey = rsaKey.public_key()
14         valid = udp.auth.validateCertificate(certificate, publicKey)
15         return jsonify({"valid": valid, "account-id":
16             attributes["account-id"]})
17     else:
18         abort(400) # missing args

```

The `udp.auth.validateCertificate` method takes a `x509.Certificate` and `rsa.RSAPublicKey` instance. The method first checks that the certificate period. If the certificate has not expired the `publicKey` can then be used to verify the certificate. If a `InvalidSignatureException` does not arise the method returns `True`. Otherwise, if either the period or verify checks fail, the method returns `False`.

```

1 def validateCertificate(certificate: x509.Certificate, publicKey:
2     rsa.RSAPublicKey) -> bool:
3     # period
4     now = datetime.datetime.now(datetime.timezone.utc)
5     if not (certificate.not_valid_before_utc <= now <=
6         certificate.not_valid_after_utc): # check in period
7         return False
8     # signature
9     try:
10         publicKey.verify( # check against publicKey
11             certificate.signature,
12             certificate.tbs_certificate_bytes,
13             aPadding.PKCS1v15(),
14             certificate.signature_hash_algorithm,
15         )
16     except InvalidSignature:
17         return False
18     return True

```

The `udp.auth.generateUserCertificate` method is updated to allow for an `Account.id` `userId` and `Account.username` `username` to be passed for embedding into the `x509.NameAttributes`.

```

1 def generateUserCertificate(key, userId: int | str | None = None, username:
2     str | None = None) -> x509.Certificate:
3     name = [
4         x509.NameAttribute(NameOID.ORGANIZATION_NAME, ORG_NAME),
5         x509.NameAttribute(NameOID.COMMON_NAME, COMMON_NAME),
6     ]
7     if userId is not None:
8         name.append(x509.NameAttribute(NameOID.USER_ID, str(userId)))
9     if username is not None:

```

```

9         name.append(x509.NameAttribute(NameOID.PSEUDONYM, username))
10    subject = issuer = x509.Name(name)
11    cert = (
12        ### omitted for clarity
13    )
14    return cert

```

The `udp.Client.validateCertificate` method can now be defined using a `requests.get` to retrieve validation from the RESTful server.

```

1 def validateCertificate(self, certificate: auth.x509.Certificate) -> bool:
2     url = f"http://{self.targetHost}:5000/auth/certificate/validate"
3     headers = {"Content-Type": "application/json"}
4     certificate = base64.encodebytes(
5         auth.getDerFromCertificate(certificate)
6     ).decode()
7     data = {"certificate": certificate}
8     try:
9         r = requests.get(url, headers=headers, data=json.dumps(data))
10        if r.status_code == 200:
11            return r.json()["valid"]
12        else:
13            return False
14    except:
15        # server unresponsive
16        return False

```

The `udp.Server.validateCertificate` is implemented as the same except for returning the account-id instead of the boolean True.

```

1 def validateCertificate(self, certificate: auth.x509.Certificate) -> bool|int:
2     # omitted for clarity
3     if r.status_code == 200:
4         return r.json()["valid"], r.json()["account-id"]
5     else:
6         return False
7     # omitted for clarity

```

5.4.4.6 RPS Demo The Rock, Paper, Scissors (`rps`) python package contains a game `rps.Server` and `rps.Client` using `udp.Server` and `udp.Client` respectively.

The choice and outcomes are defined in the package `__init__` allowing both the `Client` and `Server` to import them.

```

1 class Choice:

```

```

2     ROCK = 0
3     PAPER = 1
4     SCISSORS = 2
5
6
7 class Outcome:
8     LOOSE = 0
9     WIN = 1
10    DRAW = 2

```

The Game attributes are defined in `game_config.yaml`.

```

1 NAME: "RPS"
2 ID: 1
3 MIN_PLAYERS: 2
4 MAX_PLAYERS: 2

```

These can then be loaded using the `yaml` package in the `__init__`.

```

1 # config
2 CONFIG_PATH = os.path.join(os.path.dirname(__file__), "game_config.yaml")
3
4 with open(CONFIG_PATH) as f:
5     config = yaml.safe_load(f)
6
7 ID = config["ID"]
8 NAME = config["NAME"]
9 MIN_PLAYERS = config["MIN_PLAYERS"]
10 MAX_PLAYERS = config["MAX_PLAYERS"]

```

5.4.4.6.1 Client

The `Client` contains a simple command line UI which guides the user through playing RPS. Once a user has inputted its choice it sends the choice (with the `RELIABLE` flag set) to the `Server` and waits for a reply. Upon receiving the outcome and scores, it displays the output to the user and waits for a new choice to be selected.

The `Client` uses the `onReceiveData` callback to receive data into the `receive` method where the data is added to a `queue.Queue` `recvQueue` after being decoded. All data is sent as `RELIABLE` default packets containing a JSON encoded payload.

```

1 def send(self, addr: tuple[str, int], data: json) -> None:
2     self.udpClient.queueDefault(
3         addr, flags=lazyFlags(flag.RELIABLE), data=self.encodeData(data)
4     )
5
6 def receive(self, addr: tuple[str, int], data: bytes) -> None:
7     self.recvQueue.put((addr, self.decodeData(data)))

```

```

8     if self.onReceiveData:
9         self.onReceiveData(addr, data)
10
11 @staticmethod
12 def encodeData(data: dict) -> bytes:
13     return json.dumps(data).encode()
14
15 @staticmethod
16 def decodeData(data: bytes) -> dict:
17     return json.loads(data.decode())

```

The `gameloop` method takes a user input choice and sends it to the Server. It then waits for the `recvQueue` to contain a reply and then updates the score and displays the results to the user. The `gameThread` is defined as `self.gameThread = Thread(name=f"{addr[1]}:Gameloop", target=self.gameloop, daemon=True)` allowing the `gameloop` to execute in its own thread.

```

1 def gameloop(self) -> None:
2     print(f"{bcolors.HEADER}\n\nRock Paper Scissors{bcolors.ENDC}")
3     try:
4         while self.isRunning:
5             choice = None
6             print("Choice R[0], P[1], S[2]: ")
7             while choice is None:
8                 try:
9                     choice = inputtimeout("", timeout=10).strip()
10                    if choice == "q":
11                        print(
12                            f"{bcolors.FAIL}Quitting. Please
13                               wait...{bcolors.ENDC}"
14                        )
15                        self.isRunning = False
16                        break
17                    choice = int(choice)
18                    if choice not in (0, 1, 2):
19                        print(
20                            f"{bcolors.FAIL}Invalid choice
21                               '{choice}'.{bcolors.ENDC}"
22                        )
23                        choice = None
24                    except ValueError:
25                        print(f"{bcolors.FAIL}Invalid choice.{bcolors.ENDC}")
26                        choice = None
27                    except KeyboardInterrupt:
28                        print(f"{bcolors.FAIL}Quitting. Please
29                               wait...{bcolors.ENDC}")

```

```

27         self.isRunning = False
28         break
29     except TimeoutOccurred:
30         if not self.isRunning:
31             break
32 if self.isRunning:
33     self.send(self.udpClient.targetAddr, {"choice": choice})
34     print("Waiting for other player...")
35     while self.isRunning:
36         try:
37             addr, data = self.recvQueue.get(timeout=QUEUE_TIMEOUT)
38             break
39         except Empty:
40             pass # check still running
41     if self.isRunning:
42         match data["outcome"]:
43             case 0:
44                 o = f"You {bcolors.FAIL}LOOSE{bcolors.ENDC}. "
45             case 1:
46                 o = f"You {bcolors.OKGREEN}WIN{bcolors.ENDC}. "
47             case 2:
48                 o = f"You {bcolors.OKCYAN}DRAW{bcolors.ENDC}. "
49             case _:
50                 o = ""
51         print(
52             f"\n{o}You Picked {data['choice']}. They picked
53                 {data['otherChoice']}. \nThe score is
54                 {data['score']['score']}: {data['otherScore']['score']}."
55         )
56         if data["outcome"] == Outcome.WIN:
57             self.score += 1
58         self.recvQueue.task_done()
59 finally:
60     self.udpClient._quit()

```

The Client utilizes the onConnect to start the gameThread.

```

1 def onConnect(self, addr: tuple[str, int]) -> None:
2     self.gameThread.start()
3     try:
4         self.udpClient.mainloop(self.quit)
5     except error.PaperClipError as e:
6         match e:
7             case error.ServerDisconnectError():
8                 print(
9                     f"{bcolors.FAIL}Server connection terminated due to

```

```

                                {error.DisconnectErrorCodes.SERVER_DISCONNECT.name}:
                                {e.args[0]}\nPlease wait while connection closes
                                gracefully...{bcolors.ENDC}"
10                                )
11                                case _:
12                                    raise e
13                                if self.gameThread.is_alive():
14                                    self.gameThread.join()
15                                return None

```

5.4.4.6.2 Server

The **Server** waits for two **Clients** to join and send their choices. The server then calculates the outcome (i.e. WIN, LOSE, DRAW) and sends this to both **Clients** along with their new scores.

The **Server** contains two static methods `evaluateWin` and `evaluatePlayerChoices` which are used to calculate the winner to two choices.

```

1 @staticmethod
2 def evaluateWin(choiceOne: int, choiceTwo: int) -> int:
3     match choiceOne:
4         case Choice.ROCK:
5             match choiceTwo:
6                 case Choice.ROCK:
7                     return Outcome.DRAW
8                 case Choice.PAPER:
9                     return Outcome.LOOSE
10                case Choice.SCISSORS:
11                    return Outcome.WIN
12                case _:
13                    raise ValueError
14        case Choice.PAPER:
15            match choiceTwo:
16                case Choice.ROCK:
17                    return Outcome.WIN
18                case Choice.PAPER:
19                    return Outcome.DRAW
20                case Choice.SCISSORS:
21                    return Outcome.LOOSE
22                case _:
23                    raise ValueError
24        case Choice.SCISSORS:
25            match choiceTwo:
26                case Choice.ROCK:
27                    return Outcome.LOOSE
28                case Choice.PAPER:

```



```

29         return Outcome.WIN
30     case Choice.SCISSORS:
31         return Outcome.DRAW
32     case _:
33         raise ValueError
34     case _:
35         raise ValueError
36
37 @staticmethod
38 def evaluatePlayerChoices(choices: list[tuple[tuple[str, int], int]]):
39     outcomes = [
40         (choices[0][0], Server.evaluateWin(choices[0][1], choices[1][1])),
41         (choices[1][0], Server.evaluateWin(choices[1][1], choices[0][1])),
42     ]
43     return outcomes

```

The `onClientJoin` and `onClientLeave` callbacks are utilized to manage the `Server`'s player record `players`. `players` takes the form `dict[tuple[str, int], dict[str, int]]` where `Client` addresses are used as a key and the values contain a dictatory of player `accountId`s and `scores`. These values are retrieved and set through getters and setter which make use of a `thread.Lock` to ensure thread-safety.

```

1 def playerJoin(self, addr: tuple[str, int], accountId: int) -> None:
2     with self.playersLock:
3         self.players[addr] = {"score": 0, "accountId": accountId}
4     if self.onClientJoin:
5         self.onClientJoin(addr, accountId)
6
7 def playerLeave(self, addr: tuple[str, int], accountId: int) -> None:
8     with self.playersLock:
9         del self.players[addr]
10    if self.onClientLeave:
11        self.onClientLeave(addr, accountId)

```

The `Server` `mainloop` method waits for `MAX_PLAYERS` to join. Once enough players have join, the method calls `getChoices` and computes the `outcomes`. The `outcomes` are then restructured for each client into a payload and the scores are updated and included in the payload. The relevant payload is then dispatched to each client. The loop then checks that it is both running and the `Server` has the appropriate number of players. If so the loop repeats.

```

1 def mainloop(self) -> None:
2     self.udpServer.startThreads()
3     try:
4         while self.isRunning:
5             if self.playerCount == MAX_PLAYERS:

```

```

6         choices = self.getChoices()
7         outcomes = self.evaluatePlayerChoices(choices)
8         replies = {}
9         for addr, outcome in outcomes:
10             replies[addr] = {
11                 "outcome": outcome,
12                 "choice": [v for k, v in choices if k == addr][0],
13                 "otherChoice": [v for k, v in choices if k !=
14                                 addr][0],
15             }
16             if outcome == Outcome.WIN:
17                 self.incrementPlayer(addr)
18         scores = self.getPlayers()
19         for addr in replies:
20             replies[addr] |= {
21                 "score": scores[addr],
22                 "otherScore": [v for k, v in scores.items() if k
23                                != addr][
24                                0
25                                ],
26             }
27         self.send(addr, replies[addr])
28
29     finally:
30         self.quit()

```

The `getChoices` method waits for inputs from all players before returning their choices.

```

1 def getChoices(self) -> list[tuple[tuple[str, int], int]]:
2     choices = {}
3     while self.isRunning:
4         try:
5             addr, data = self.recvQueue.get(timeout=QUEUE_TIMEOUT)
6             choices[addr] = data["choice"]
7             if len(choices) == 2:
8                 choices = [(addr, choice) for addr, choice in
9                             choices.items()]
10                self.recvQueue.task_done()
11                return choices
12            except Empty:
13                pass # check still running

```

5.4.4.7 Client The `Client` python package contains a simple command line UI that handles communication (including providing the reliant authentication) to the RESTful server using the `requests` package. Once a user creates a new or joins a Lobby the `Client` creates

the relevant game client and connects to the game server. If a user exits a game, it is returned to the `Client` command line UI.

5.4.4.7.1 Authentication

On `Client` initialization the `Client` uses the provided username and password to retrieve the session token.

```
1 @staticmethod
2 def getToken(username: str, password: str) -> str:
3     url = SERVER_URL + "/auth/token"
4     r = requests.get(url, auth=(username, password))
5     assert r.status_code == 200, r
6     return r.json()["token"]
```

The `Client` then uses `requests.auth.HTTPBasicAuth` for the rest of the session communications.

```
1 class Client:
2     def __init__(self, username: str, password: str, token: str | None =
        None) -> None:
3         self.username = username
4         self.password = password
5         self.gameClient = None
6         self.token = (
7             token if token is not None else self.getToken(self.username,
                self.password)
8         )
9         self.auth = HTTPBasicAuth(self.token, "")
10        self.getKey(password.encode())
```

The `Client` maps all of the RESTful endpoints to methods containing the relevant endpoint and request (with auth).

5.4.4.7.2 UI

The user is initially greeted with the options to either log-in or create an account.

```
Lobby.
1. Login
2. Register
3. Quit
: 1

Login.
Username: Mario
Password: ItsAMe123

Main Menu
Hello Mario.

1. Manage friends
2. See available games
3. Start or join a lobby
4. Quit
: 1

Friends.
Friend list:
    1. Peach

1. Add New Friend
2. Remove Friend
3. Return to Main Menu
: 3

Main Menu
Hello Mario.

1. Manage friends
2. See available games
3. Start or join a lobby
4. Quit
: 
```

Figure 4: example output of client with logging and friends

The user is guided through various text menus allowing them to perform various task including:

- viewing and managing account Friends
- view all Games
- create or join a Lobby

```

Main Menu
Hello Mario.

1. Manage friends
2. See available games
3. Start or join a lobby
4. Quit
: 3

Lobby.

1. Matchmaking
2. See Friends' Lobbies
3. Join Lobby
4. Create Lobby
5. Return to Main Menu
: 1

Matchmaking.

Available Games:
    1. RPS
Game: rps

Joining Lobby '1'

Rock Paper Scissors
Choice R[0], P[1], S[2]:
0
Waiting for other player...

```

Figure 5: example output of client with matchmaking and game client creation

5.4.4.7.3 Matchmaking

The Lobbies menu includes the option for matchmaking. The `_matchmaking` method first attempts to find an available Lobby. If this fails, the method creates a new Lobby. The relevant game Client is then created and connected to the game Server.

```

1 def _matchmaking(client: Client):
2     print(f"{bcolors.HEADER}\nMatchmaking.{bcolors.ENDC}")
3     game = _gameInput(client)
4     if game is None:
5         return None
6     try:
7         lobby = client.findLobby(gameName=game)

```

```

8     except AssertionError:
9         lobby = client.createLobby(gameName=game)
10    time.sleep(1)
11    client.join(lobby["lobby-id"])
12    return None

```

The join method handles creating the relevant game Client and joining the Lobby using the provided lobbyId.

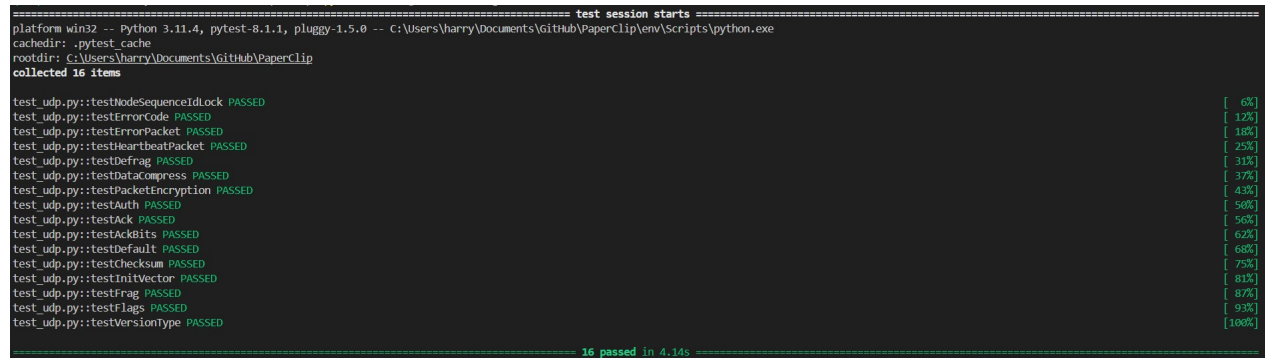
```

1 def join(self, lobbyId: int) -> None:
2     print(f"\n{bcolors.WARNING}Joining Lobby '{lobbyId}'{bcolors.ENDC}")
3     data = self.getLobby(lobbyId)
4     if data["lobby-addr"] is not None:
5         match data["game-id"]:
6             case 1:
7                 self.gameClient = RpsClient(
8                     (TCP_HOST, C_PORT),
9                     data["lobby-addr"],
10                    rsaKey=self.key,
11                    userId=self.id,
12                    username=self.username,
13                )
14                self.gameClient.connect()
15            case _:
16                raise ValueError(f"Unknown gameId {data['game-id']}")

```

5.5 Tests

The pytest module was used to define serval test.



```

===== test session starts =====
platform win32 -- Python 3.11.4, pytest-8.1.1, pluggy-1.5.0 -- C:\Users\harry\Documents\Github\PaperClip\env\Scripts\python.exe
cachedir: .pytest_cache
rootdir: C:\Users\harry\Documents\Github\PaperClip
collected 16 items

test_udp.py::testNodeSequenceIdLock PASSED [ 6%]
test_udp.py::testErrorCode PASSED [ 12%]
test_udp.py::testErrorPacket PASSED [ 18%]
test_udp.py::testHeartbeatPacket PASSED [ 25%]
test_udp.py::testDefrag PASSED [ 31%]
test_udp.py::testDataCompress PASSED [ 37%]
test_udp.py::testPacketEncryption PASSED [ 43%]
test_udp.py::testAuth PASSED [ 50%]
test_udp.py::testAck PASSED [ 56%]
test_udp.py::testAckRits PASSED [ 62%]
test_udp.py::testDefault PASSED [ 68%]
test_udp.py::testChecksum PASSED [ 75%]
test_udp.py::testInitVector PASSED [ 81%]
test_udp.py::testFrag PASSED [ 87%]
test_udp.py::testFlags PASSED [ 93%]
test_udp.py::testVersionType PASSED [100%]

===== 16 passed in 4.14s =====

```

Figure 6: example output of pytest -v

6 Conclusion

6.1 Project Objective

The main objectives for this project where:

1. Create a scalable system for managing user accounts and inter-account interactions including matchmaking and friends.
2. Create a custom UDP protocol than implement key features required for game communication missing from vanilla UDP. These include reliability and security.

Both objectives have been met.

6.1.1 Objective One The `server` package provides a method to manipulate the various account and inter-account rows from the database which in turn provides scalable storage of data. It additionally provides the necessary information required to find or create `Lobbys`.

The `client` package provides a user-friendly interface for interactions with the `Server`. Additionally, using the information provided by the `Server`, is able to matchmaking by joining available lobbies, and only creating new ones where needed/requested.

6.1.2 Objective Two The `udp` package defines the required UDP packets and implements their key features in the `udp.Client` and `udp.Server` communication.

6.1.3 Secondary Objectives

- Turn-based game demo
 - Implemented in the `rps` package.
- Real-time game demo
 - Omitted due to time constraints.
- Large packet size limit
 - Implemented in the `udp` package with the use of the `FRAG` flag.
- Authenticated Communication
 - Implemented in the `udp` package with the use of the `AUTH` packet containing a `X.509 Certificate`.
- Secure Communication
 - Implemented in the `udp` package with the use of the `ENCRYPT` flag.
- Reliable Communication
 - Implemented in the `udp` package with the use of the `RELIABLE` flag.
- Lightweight Communication
 - Implemented with the custom UDP protocol defined in `udp` which negates TCP's overhead.
- Modularity
 - Implemented in the project's overall structure such that a game can be defined in a package.

6.2 Feature Evaluation

6.2.1 Reflection on Handshake Finished The handshake **Finished** is generated by calculating a **Hash-based Message Authentication Code (HMAC)** using the session key. Though the **EC** keys used to generate the session key are ephemeral, the handshake exchange is still vulnerable to man-in-the-middle attacks. This could be mitigated by including some random int in each **AUTH** packet which is, in turn, included in the **Finished** **HMAC** calculation.

6.2.2 Port Selection and Assignment The **Server** is responsible for selecting and assigning ports to any game servers it creates. The method for selection is rather rudimentary as the **Server** will just iterate from a given starting value indefinitely during run time. There are, therefore, no additional checks to validate either the availability (i.e. is the port is free) or suitability (e.g. within some reasonable range) of the port selected. A more advanced system could be implemented allowing for ports used by the **Server** to be monitored and released (e.g. by **LobbyHandler**) after use such that they could be reused.

6.2.3 Packet ERRORS Though **Packer** **ERRORs** are raised during run time and then sent to the packet author, there is no mechanism in place to resend the packet. This would require a greater overhead as all recent packets would need to be stored such that they could be retired if a packet error occurred. This is ultimately outside of the scope of this project.

6.2.4 Automatic Game Importing Though each game can be defined modularly using a package, the relevant game **Server** and **Client** must be explicitly imported into both the **server** and **client** packages. This limitation is largely due to the nature of **Python** imports and package structure. Some attempt could be made with the use of the underlying methods used by the **import** keyword but this would likely generate messy ‘un-pythonic’ code.

6.2.5 Congestion Control Currently, the custom **udp** implements no congestion control. The use of a **Round-Time-Time (RTT)** could be used to artificially rate-limit a **Node**’s send rate. The **HEARTBEAT** packet contains an unused **data** field which would be a suitable candidate for transferring additional information about a **Node**’s network status.

6.2.6 Features Cut for Time The **Scores** model and the highscore tracking functionality was ultimately abandoned due to time constraints. This feature could be implemented by returning the score values (already tracked by **Lobby** instances) and committing to the database.

Additionally, the stretch goal for a real-time game demo was never implemented. This would have been a more suitable demonstration of the efficiency of the **Packet** implementation but required far greater overhead for design and implementation than its turn-based counterpart.

7 Glossary

- Address, `addr`: a tuple containing an IP address and port number in the form (IP, port).
- Advanced Encryption Standard (AES): AES is a symmetric encryption algorithm used to secure sensitive data.
- Application Programming Interface (API): An API is a software interface allowing for two or more computer programs or components to communicate.
- Asymmetric Encryption: Asymmetric Encryption is a cryptographic technique where a pair of keys (a public and private key) are used for encryption and decryption. Messages encrypted with a given key pair's public key can only be decrypted with the respective private key (and visa versa).
- CRC-32: CRC-32 is a 32-bit checksum based on the data content of a given object.
- Checksum: A checksum is a value calculated from a data packet using some mathematical algorithm such that the same data input always yields the same checksum. The checksum is used to validate data integrity after transmission to ensure no corruption has occurred.
- Constant (CONST): A constant is a value that cannot be changed during the execution of a program.
- Daemon: A daemon is a background process. Daemons will typically run continuously until stopped and operate silently, without direct user interaction.
- Diffie-Hellman (DH): Diffie-Hellman is a key exchange algorithm used to create a shared secret key between two parties. It allows the parties to agree on a secret key without the need to explicitly share the key over some communication channel.
- Distinguished Encoding Rules (DER): DER is a standard for encoding data structures into binary form for transmission or storage. It is a more restrictive variant of BER that ensures, among other things, that the shortest possible length encoding is used.
- Docker: Docker is a platform for creating and managing containers. Containers are lightweight, standalone and portable environments that typically package an application with its required dependencies.
- Elliptic-curve (EC): Elliptic-curve is a form of asymmetric cryptography based on the structure of elliptic curves over finite fields.
- Elliptic-curve Diffie-Hellman (ECDH): ECDH is a form of Diffie-Hellman that utilizes Elliptic-curve keys.
- Elliptic-curve Diffie-Hellman Ephemeral (ECDHE): ECDHE is the ephemeral form of ECDH. ECDHE provides forward security by generating a new ephemeral key pair for each session thus yielding a unique secret key.

- Entity-Relationship Diagram (ERD): An ERD is a UML representation of data models, their attributes and the relationships between them in a database.
- Enum: An enum is a data type that consists of a set of named values with a distinct constant value.
- Exception: An exception is an event that occurs during the execution of a program that disrupts the typical flow of instructions (i.e. an error).
- Flask: Flask is a micro-web framework package for Python.
- Git: Git is a source control system that tracks changes (and blame) for a given set of computer files.
- GitHub: GitHub is a developer platform providing cloud storage and management of Git projects.
- Global Interpreter Lock (GIL): The GIL is a mechanism used to ensure that only one thread executes bytecode at any given time.
- Hash-based Message Authentication Code (HMAC): HMAC is a type of message authentication code that involves the combination of a cryptographic hash function and a secret key to generate a unique code.
- Hypertext Transfer Protocol (HTTP): HTTP is a networking, application layer, protocol for distributed hypermedia information systems. HTTP provides the foundation for data communication on the World Wide Web.
- JSON Web Token (JWT): JWT is a standard for creating data with, optional signature and/or encryption, whose payload contains some JSON value typically asserting some number of claims (e.g. username is foo).
- JavaScript Object Notation (JSON): JSON is a data interchange format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs. Despite its namesake, JSON is platform agnostic.
- Mutex: A mutex is a programming construct used to prevent multiple threads from accessing a shared resource simultaneously.
- MySQL: MySQL is a relational database management system (RDBMS).
- Private Key: The private key is one-half of the asymmetric key pair and is kept secret.
- Public Key: The public key is one-half of the asymmetric key pair and is freely distributed.
- Python: Python is a high-level, general-purpose programming language.
- Queue: A queue is a First-In-First-Out (FIFO) data structure.
- Race Condition: In concurrent programming, a race condition arises when the outcome of a program is affected by the unpredictable timing of multiple threads accessing a shared resource.

- **Rivest-Shamir-Adleman (RSA):** RSA is a form of asymmetric cryptography that utilizes the factorization of large integers.
- **SQLAlchemy:** SQLAlchemy is a SQL toolkit and object-relational-mapper (ORM) for Python.
- **Session Key:** A session key is a temporary cryptographic key used for encryption and decryption during a specific communication session between two parties.
- **Symmetric Encryption:** Symmetric Encryption is a cryptographic technique where the same key is used for both encryption and decryption of data.
- **Thread:** A thread is a lightweight process that can execute independently and concurrently with other threads in the same process.
- **Transmission Control Protocol (TCP):** TCP is a networking, transport layer, protocol that enables connection-based communication. It provides reliable, ordered and error-checked packet transmission.
- **Try/Except Block:** A try/except block is a mechanism used for exception handling. Any exceptions raised during the try block will be checked against the exception(s) defined in the except block. If the raised exception matches any of the defined exceptions, the exception is caught and not raised any further.
- **Unified Modeling Language (UML):** UML is a general-purpose visual modelling language for use with system design.
- **User Datagram Protocol (UDP):** UDP is a networking, transport layer, protocol that enables connectionless communication. It prioritizes speed over reliability and, as such, is unreliable and unordered.
- **Visual Studio Code (VS Code):** Visual Studio Code is a free, open-source code editor.
- **X.509:** X.509 is a standard format for certificates allowing for signing with an asymmetric key. A X.509 will typically also contain some information about the identity of the author.
- **Zlib:** Zlib is a software library used for data compression.

8 References

Capcom, D. (2008) ‘Street fighter IV’. Capcom.

Eddy, W. (2022) *Transmission Control Protocol (TCP)*. 9293. RFC 9293; RFC Editor. Available at: <https://doi.org/10.17487/RFC9293>.

Fiedle, G. (2008a) *Reliability and congestion avoidance over UDP*. Available at: https://web.archive.org/web/20180823011635/https://gafferongames.com/post/reliability_ordering_and_congestion_avoidance_over_udp/ (Accessed: 8 May 2024).

Fiedle, G. (2008b) *UDP vs. TCP*. Available at: https://web.archive.org/web/20180823015049/https://gafferongames.com/post/udp_vs_tcp/ (Accessed: 8 May 2024).

Firaxis Games (2001) ‘Sid meier’s civilization III’. Infogrames Interactive.

Howard Wexler (1974) ‘Connect four’. Milton Bradley.

id Software (1996) ‘Quake’. GT Interactive.

Internet Engineering Task Force (2006) *X.509 public key infrastructure certificate and certificate revocation list (CRL) profile*. Internet Engineering Task Force (IETF). Available at: <https://www.rfc-editor.org/rfc/rfc5280.html>.

Jan Algermissen (2010) *Classification of HTTP APIs*. algermissen.io. Available at: http://algermissen.io/classification_of_http_apis.html (Accessed: 29 January 2023).

Mitsuo Heijo (2017) ‘Inputtimeout’, *GitHub repository*. GitHub. Available at: <https://github.com/johejo/inputtimeout>.

Postel, J. (1980) *User Datagram Protocol*. 768. RFC 768; RFC Editor. Available at: <https://doi.org/10.17487/RFC0768>.

Rescorla, E. (2018) *The Transport Layer Security (TLS) Protocol Version 1.3*. 8446. RFC 8446; RFC Editor. Available at: <https://doi.org/10.17487/RFC8446>.

Ronald L. Rivest, L.M.A., Adi Shamir (1978) ‘A method for obtaining digital signatures and public-key cryptosystems’, *Communications of the ACM*, 21(2), pp. 120–126. Available at: <https://doi.org/10.1145/359340.359342>.

Roy Thomas Fielding (2000) *Architectural styles and the design of network-based software architectures*. PhD thesis. University of California, Irvine. Available at: <https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.

Salzman, L. (2024) ‘ENet’, *GitHub repository*. GitHub. Available at: <https://github.com/lsalzman/enet>.

ValveSoftware (2022) ‘GameNetworkingSockets’, *GitHub repository*. GitHub. Available at: <https://github.com/ValveSoftware/GameNetworkingSockets>.

van Oortmerssen, W. (2005) ‘Cube’.

Wang, Y., Ramamurthy, B. and Zou, X. (2006) ‘The performance of elliptic curve based group diffie-hellman protocols for secure group communication over ad hoc networks’, in *2006 IEEE international conference on communications*, pp. 2243–2248. Available at: <https://doi.org/10.1109/ICC.2006.255104>.

wolfSSL (2019) *TLS 1.3 performance analysis – full handshake*. Available at: <https://www.wolfssl.com/tls-1-3-performance-part-2-full-handshake-2/>.

9 Appendices

9.1 Project Definition Document

Cover Sheet

- PaperClip: A backend networking and account management solution for game servers.
- Author: Harry Whitehorn - harry.whitehorn@city.ac.uk
- Course: BSc Computer Science
- Consultant: Stephanie Wilson - s.m.wilson@city.ac.uk
- Proposed by: Harry Whitehorn
- Proprietary Interests: N/A
- Word Count: 697

Proposal

Problem to be solved Networking and user interaction is a basic requirement for many online video games. This can require a large amount of time and cost from developers. This project will aim to provide a lightweight solution to both of these elements that can then be easily applied to a game-server.

There are a variety of different methods for handling game-time networking such as delay-based and rollback. The former is more suited to slow or turn based games due to the nature of an implicit delay as users synchronize. Fast-paced games such as shooters or fighting games rely on quick interactions meaning that delay-based networking is not suitable. Rollback works by predicting and then updating the inputs of non-local players.



Figure 7: Screenshot of *Civ II* (1996)

MicroProse (1996) *Civ II* (1996) is a turn based game suitable for delay-based.



Figure 8: Screenshot of *Super Street Fighter II Turbo HD Remix* (2008)

Backbone Entertainment (2008) *Super Street Fighter II Turbo HD Remix* was one of the first games released with rollback.

Project objectives The primary objective of this project is to create a lightweight networking and user account solution. This can better be described with two main goals:

1. This project shall manage connections between a game-sever and game-client. This will include handling both matchmaking and in-play communications.
2. This project shall manage storing and retrieving non-volatile information from a database sever. This will include user information and, therefore, handling account validation as well as leaderboards.
3. Create a variety of small and simple games to implement the networking and account features.

Further objectives include:

- A ELO/rank based system for matching players of a equal skill level.
- Account interaction such as friends and private lobbies.
- Customable bot accounts for relevant use in multiplayer environments.
- Ensuring that the solution is lightweight, reliable and platform-agnostic.
- Dockerize.

Beneficiaries The primary beneficiaries include:

- Me, due to learning network intricacies.
- Independent or small game development teams, who might lack either the time or resources to develop custom handling for multiplayer.

Work Plan The project will be created primarily to work with the godot game engine. Additional game engine support will be implemented and testing if there is ample time but this is not expected. The best language to use will be part of the initial research, but python is a likely candidate. Additionally, the database management system best suited will be part of the research but some potential candidates include SQLite and MariaDB.

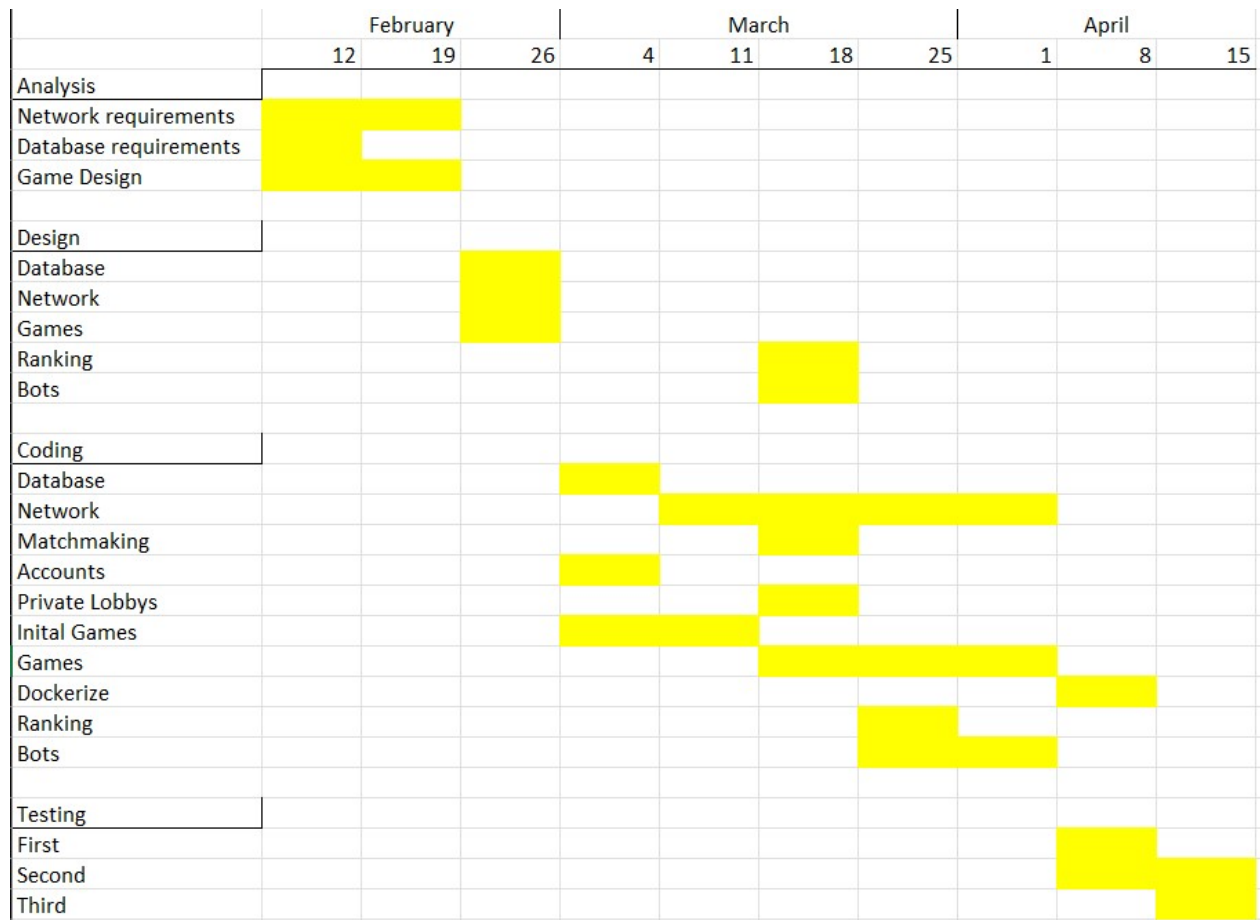


Figure 9: workplan

Risks

Objectiv	Risk	Severity	Score	Risks	Actions
Network2	5	10	Vulnerabilities	Ensure that the system is not vulnerable to any malicious interaction. Including ‘cheating’.	
Account4	4	4	GDPR	Limit type of stored data and where data is relevant unsure stored in accordance to GDPR.	
Safeguar2ling	4	8	User safety	Limit interactions between user accounts.	
Develop5ment	5	25	Time	Ensure that progress is made in accordance with work plan and actively update when system requirements change.	

Objective	Risk	Severity	Score	Risks	Actions
Testing	3	4	12	Engagement	Ensure in advance that participants are willing and have the appropriate time and resources to complete required testing.
	4	4	16	Time	Ensure that time is left to testing can be completed. (See development actions.)
Matchmaking	1		3	Developing appropriate ELO/rank system may prove to be too complex.	ELO/rank system can be dropped in favour of random matchmaking if the former cannot be developed within the time constraints.
Bots	4	2	8	Developing appropriate bot system may prove to be too complex.	Bot system can be dropped if it cannot be finished in time.

Research Ethics Checklist

See `Ethics_Review_Form.doc`

References

1. MicroProse, 1996, *Civilization II*, MicroProse
2. DASHBot, (2010). *CivII 01.png*. [image online] Wikipedia. Available at: https://en.wikipedia.org/wiki/File:CivII_01.png [Accessed 10 Feb. 2024].
3. Backbone Entertainment, 2008, *Super Street Fighter II Turbo HD Remix*, Capcom
4. Gamescore Blog, (2008). *Super Street Fighter II Turbo HD Remix 12*. [image online] Available at: <https://www.flickr.com/photos/gamerscore/3058520175> [Accessed 10 Feb. 2024].

9.2 Deployment Guide

Requirements Requires: `python 3.11+`, `mysql server`.

Note This projects makes use of ANSI color prints.

This feature is shipped with **Windows 10 - Build 16257** and later but may require enabling. Instructions for registry to enable global default..

Failure to enable ANSI color prints may result in color codes (e.g. "\033[94m") being printed to the terminal.

mysql A mysql server must be set up. The connection path must then be set as `SQLALCHEMY_DATABASE_URI` in the `.env`. For convenience, the following is a example docker-compose to set up a mysql server and adminer. This is the method use in testing thus the example path set in the `.env`.

```
1 # docker-compose.yaml
2 version: '3.1'
3
4 services:
5
6   adminer:
7     image: adminer
8     ports:
9       - 8080:8080
10
11   db:
12     image: mysql:5.6
13     environment:
14       MYSQL_ROOT_PASSWORD: root
15     ports:
16       - 3306:3306
```

Python

Setup Optional: `python -m venv env` and activate

Install packages: `pip install -r requirements.txt`

Env

Example `.env`, **must** be place in app root

```
1 # udp
2 S_HOST=127.0.0.1
3 S_PORT=2024
4 C_HOST=127.0.0.1
5 C_PORT=2025
6 ## node
7 SOCKET_BUFFER_SIZE = 1024
8 SEND_SLEEP_TIME = 0.1
9 QUEUE_TIMEOUT = 10
10 SOCKET_TIMEOUT = 20
11 ## server
12 HEARTBEAT_MAX_TIME = 120
13 HEARTBEAT_MIN_TIME = 30
14 MAX_CLIENTS
15 ## auth
```

```

16 ORG_NAME = Paperclip
17 COMMON_NAME = 127.0.0.1
18 ## utils
19 MAX_FRAGMENT_SIZE = 988
20
21 # client
22 TCP_PORT = 5000
23
24 # app
25 FLASK_APP=server
26 PRUNE_TIME = 58
27 SECRET_KEY = MyVerySecretKey
28 SQLALCHEMY_DATABASE_URI = mysql://root:root@localhost:3306/paperclip
29
30 # debug
31 DEBUG = True

```

Run Server: `python -m flask run`

Client: `python -m client` or `python -m client offset` (where *offset* is some `int` such that `C_PORT` (from `.env`) becomes `C_PORT+=offset`)

Tests: `pytest -v` (Note: may take some time with no output due to testing of thread locks)

9.3 Package

9.3.1 udp

```

1 # udp.__init__
2 import logging
3 import os
4 import sys
5
6 import dotenv
7
8 __version__ = 0
9
10 dotenv.load_dotenv(".env")
11 S_HOST = os.environ.get("S_HOST")
12 S_PORT = int(os.environ.get("S_PORT"))
13 C_HOST = os.environ.get("C_HOST")
14 C_PORT = int(os.environ.get("C_PORT"))
15 # node
16 SOCKET_BUFFER_SIZE = int(os.environ.get("SOCKET_BUFFER_SIZE"))

```

```

17 SEND_SLEEP_TIME = float(os.environ.get("SEND_SLEEP_TIME"))
18 QUEUE_TIMEOUT = int(os.environ.get("QUEUE_TIMEOUT"))
19 SOCKET_TIMEOUT = int(os.environ.get("SOCKET_TIMEOUT"))
20 # server
21 HEARTBEAT_MAX_TIME = int(os.environ.get("HEARTBEAT_MAX_TIME"))
22 HEARTBEAT_MIN_TIME = int(os.environ.get("HEARTBEAT_MIN_TIME"))
23 MAX_CLIENTS = (
24     int(os.environ.get("MAX_CLIENTS"))
25     if os.environ.get("MAX_CLIENTS") is not None
26     else float("inf")
27 )
28 # auth
29 ORG_NAME = os.environ.get("ORG_NAME")
30 COMMON_NAME = os.environ.get("COMMON_NAME")
31 # utils
32 MAX_FRAGMENT_SIZE = int(os.environ.get("MAX_FRAGMENT_SIZE"))
33
34
35 class bcolors:
36     HEADER = "\033[95m"
37     OKBLUE = "\033[94m"
38     OKCYAN = "\033[96m"
39     OKGREEN = "\033[92m"
40     WARNING = "\033[93m"
41     FAIL = "\033[91m"
42     ENDC = "\033[0m"
43     BOLD = "\033[1m"
44     UNDERLINE = "\033[4m"
45
46
47 class ColorFilter(logging.Filter):
48     colorCodes = [
49         getattr(bcolors, attr) for attr in dir(bcolors) if not
            attr.startswith("__")
50     ]
51
52     def filter(self, record: logging.LogRecord) -> bool:
53         for color in self.colorCodes:
54             record.msg = record.msg.replace(color, "")
55         return True
56
57
58 logger = logging.getLogger(__name__)
59 logger.setLevel(logging.DEBUG)
60

```

```

61 printHandler = logging.StreamHandler(sys.stdout)
62 printHandler.setLevel(logging.INFO)
63 printHandler.setFormatter(
64     logging.Formatter(f"{bcolors.OKBLUE}%(threadName)s{bcolors.ENDC} -
        %(message)s")
65 )
66 logger.addHandler(printHandler)
67
68 fileHandler = logging.FileHandler("paperclip.log")
69 fileHandler.setLevel(logging.DEBUG)
70 fileHandler.addFilter(ColorFilter())
71 fileHandler.setFormatter(
72     logging.Formatter("%(asctime)s - %(levelname)s - %(threadName)s -
        %(message)s")
73 )
74 logger.addHandler(fileHandler)

```

```

1  # upd.__main__
2  from . import client, server
3
4
5  def runServer():
6      s = server.Server((S_HOST, S_PORT))
7      s.startThreads()
8      return s
9
10
11 def runClient():
12     c = client.Client((C_HOST, C_PORT), (S_HOST, S_PORT))
13     c.connect()
14     return c
15
16
17 if __name__ == "__main__":
18     import time
19
20     from . import C_HOST, C_PORT, S_HOST, S_PORT
21
22     s = runServer()
23     time.sleep(1)
24     c = runClient()
25     time.sleep(1)
26     x = None

```

```

27     x = input("> ")
28     while x != "END":
29         c.queueDefault(data=x.encode())
30         x = input("> ")
31     c.isRunning.clear()
32     time.sleep(1)
33     s.isRunning.clear()
34     time.sleep(1)
35     print("END")

```

```

1  # udp.auth
2  import datetime
3  import os
4
5  from cryptography import x509
6  from cryptography.exceptions import InvalidSignature
7  from cryptography.hazmat.primitives import hashes, hmac, padding,
    serialization
8  from cryptography.hazmat.primitives.asymmetric import ec, rsa
9  from cryptography.hazmat.primitives.asymmetric import padding as aPadding
10 from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
11 from cryptography.hazmat.primitives.kdf.hkdf import HKDF
12 from cryptography.x509.oid import NameOID
13
14 from . import COMMON_NAME, ORG_NAME
15
16
17 def generateRsaKey() -> rsa.RSAPrivateKey:
18     key = rsa.generate_private_key(
19         public_exponent=65537,
20         key_size=2048,
21     )
22     return key
23
24
25 def getDerFromRsaPrivate(key: rsa.RSAPrivateKey, password: bytes) -> bytes:
26     der = key.private_bytes(
27         encoding=serialization.Encoding.DER,
28         format=serialization.PrivateFormat.PKCS8,
29         encryption_algorithm=serialization.BestAvailableEncryption(password),
30     )
31     return der
32

```

```

33
34 def getRsaPrivateFromDer(data: bytes, password: bytes) -> rsa.RSAPrivateKey:
35     key = serialization.load_der_private_key(data, password=password)
36     return key
37
38
39 def getDerFromRsaPublic(key: rsa.RSAPublicKey) -> bytes:
40     der = key.public_bytes(
41         encoding=serialization.Encoding.DER,
42         format=serialization.PublicFormat.SubjectPublicKeyInfo,
43     )
44     return der
45
46
47 def getRsaPublicFromDer(data: bytes) -> rsa.RSAPublicKey:
48     key = serialization.load_der_public_key(data)
49     return key
50
51
52 def generateUserCertificate(
53     key, userId: int | str | None = None, username: str | None = None
54 ) -> x509.Certificate:
55     name = [
56         x509.NameAttribute(NameOID.ORGANIZATION_NAME, ORG_NAME),
57         x509.NameAttribute(NameOID.COMMON_NAME, COMMON_NAME),
58     ]
59     if userId is not None:
60         name.append(x509.NameAttribute(NameOID.USER_ID, str(userId)))
61     if username is not None:
62         name.append(x509.NameAttribute(NameOID.PSEUDONYM, username))
63     subject = issuer = x509.Name(name)
64     cert = (
65         x509.CertificateBuilder()
66         .subject_name(subject)
67         .issuer_name(issuer)
68         .public_key(key.public_key())
69         .serial_number(x509.random_serial_number())
70         .not_valid_before(datetime.datetime.now(datetime.timezone.utc))
71         .not_valid_after(
72             datetime.datetime.now(datetime.timezone.utc) +
73             datetime.timedelta(days=1)
74         )
75         .add_extension(
76             x509.SubjectAlternativeName([x509.DNSName("localhost")]),
77             critical=False,

```

```

77         )
78         .sign(key, hashes.SHA256())
79     )
80
81     return cert
82
83
84 def getUserCertificateAttributes(certificate: x509.Certificate) -> list:
85     accountId = certificate.subject.get_attributes_for_oid(NameOID.USER_ID)
86     accountId = accountId[0].value if len(accountId) > 0 else None
87     username = certificate.subject.get_attributes_for_oid(NameOID.PSEUDONYM)
88     username = username[0].value if len(username) > 0 else None
89     return {"account-id": accountId, "username": username}
90
91
92 def validateCertificate(
93     certificate: x509.Certificate, publicKey: rsa.RSAPublicKey
94 ) -> bool:
95     # period
96     now = datetime.datetime.now(datetime.timezone.utc)
97     if not (certificate.not_valid_before_utc <= now <=
98             certificate.not_valid_after_utc):
99         return False
100    # signature
101    try:
102        publicKey.verify(
103            certificate.signature,
104            certificate.tbs_certificate_bytes,
105            aPadding.PKCS1v15(),
106            certificate.signature_hash_algorithm,
107        )
108    except InvalidSignature:
109        return False
110    return True
111
112 def generateEcKey() -> ec.EllipticCurvePrivateKey:
113     key = ec.generate_private_key(ec.SECP384R1())
114     return key
115
116
117 def getDerFromPublicEc(publicKey: ec.EllipticCurvePublicKey) -> bytes:
118     ecDer = publicKey.public_bytes(
119         encoding=serialization.Encoding.DER,
120         format=serialization.PublicFormat.SubjectPublicKeyInfo,

```



```

121 )
122     return ecDer
123
124
125 def getPublicEcFromDer(publicKeyDer: bytes) -> ec.EllipticCurvePublicKey:
126     ec_ = serialization.load_der_public_key(publicKeyDer)
127     return ec_
128
129
130 def getDerFromCertificate(certificate: x509.Certificate) -> bytes:
131     return certificate.public_bytes(serialization.Encoding.DER)
132
133
134 def getCertificateFromDer(certificateDer: bytes) -> x509.Certificate:
135     return x509.load_der_x509_certificate(certificateDer)
136
137
138 def generateSessionKey(
139     localKey: ec.EllipticCurvePrivateKey, peerKey: ec.EllipticCurvePublicKey
140 ) -> bytes:
141     sessionSecret = localKey.exchange(ec.ECDH(), peerKey)
142     sessionKey = HKDF(
143         algorithm=hashes.SHA256(), length=32, salt=None, info=b"handshake
144         data"
145     ).derive(sessionSecret)
146     return sessionKey
147
148 def encryptBytes(cipher: Cipher, rawBytes: bytes, autoPad=True) -> bytes:
149     if autoPad:
150         padder = padding.PKCS7(algorithms.AES.block_size).padder()
151         rawBytes = padder.update(rawBytes) + padder.finalize()
152     encryptor = cipher.encryptor()
153     encryptedBytes = encryptor.update(rawBytes) + encryptor.finalize()
154     return encryptedBytes
155
156
157 def decryptBytes(
158     cipher: Cipher, encryptedBytes: bytes, autoUnpad: bool = True
159 ) -> bytes:
160     decryptor = cipher.decryptor()
161     decryptedBytes = decryptor.update(encryptedBytes) + decryptor.finalize()
162     if autoUnpad:
163         unpadder = padding.PKCS7(algorithms.AES.block_size).unpadder()
164         decryptedBytes = unpadder.update(decryptedBytes) + unpadder.finalize()

```

```

165     return decryptedBytes
166
167
168 def generateInitVector() -> bytes:
169     return os.urandom(16)
170
171
172 def generateCipher(
173     sessionKey: bytes, iv: bytes = generateInitVector()
174 ) -> tuple[Cipher, bytes]:
175     cipher = Cipher(algorithms.AES(sessionKey), modes.CBC(iv))
176     return cipher, iv
177
178
179 def generateFinished(sessionKey: bytes, finishedLabel: bytes, messages:
    bytes):
180     hashValue = hashes.Hash(hashes.SHA256())
181     hashValue.update(messages)
182     hashValue = hashValue.finalize()
183
184     prf = hmac.HMAC(sessionKey, hashes.SHA256())
185     prf.update(finishedLabel)
186     prf.update(hashValue)
187     prf = prf.finalize()
188
189     return prf

```

```

1  # udp.client
2  import base64
3  import json
4  import socket
5
6  import requests
7  from cryptography.hazmat.primitives.asymmetric.rsa import RSAPrivateKey
8
9  from udp.error import Major, Minor
10
11 from . import auth, bcolors, error, logger, node, packet
12
13
14 class Client(node.Node):
15     targetAddr: tuple[str, int]
16     rsaKey: RSAPrivateKey

```

```

17     onConnect: None
18     onDisconnect: None
19
20     def __init__(
21         self,
22         addr,
23         targetAddr,
24         rsaKey: RSAPrivateKey | None = None,
25         accountId: int | str | None = None,
26         username: str | None = None,
27         onConnect=None,
28         onDisconnect=None,
29         onReceiveData=None,
30     ) -> None:
31         self.targetAddr = targetAddr
32         self.rsaKey = rsaKey if rsaKey is not None else auth.generateRsaKey()
33         self.onConnect = onConnect
34         self.onDisconnect = onDisconnect
35         s = socket.socket(type=socket.SOCK_DGRAM)
36         super().__init__(
37             addr,
38             cert=auth.generateUserCertificate(self.rsaKey, accountId,
39                                               username),
40             accountId=accountId,
41             socket=s,
42             onReceiveData=onReceiveData,
43         )
44         self.regenerateEcKey()
45         self.bind(self.addr)
46
47     @property
48     def targetHost(self) -> str | None:
49         return self.targetAddr[0] if self.targetAddr is not None else None
50
51     @property
52     def targetPort(self) -> int | None:
53         return self.targetAddr[1] if self.targetAddr is not None else None
54
55     def queueDefault(
56         self,
57         addr: tuple[str, int] = None,
58         flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
59         data: bytes | None = None,
60     ) -> None:
61         return super().queueDefault(self.targetAddr, flags=flags, data=data)

```

```

61
62 def queueACK(
63     self,
64     addr: tuple[str, int] = None,
65     ackId: int = None,
66     flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
67     data: bytes | None = None,
68 ) -> None:
69     return super().queueACK(self.targetAddr, ackId, flags=flags,
70                             data=data)
71
72 def queueError(
73     self,
74     addr: tuple[str, int] = None,
75     major: Major | int = 0,
76     minor: Minor | int = 0,
77     flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
78     data: bytes | None = None,
79 ) -> None:
80     return super().queueError(self.targetAddr, major, minor, flags, data)
81
82 def queueDisconnect(
83     self,
84     addr: tuple[str, int] = None,
85     flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
86     data: bytes | None = None,
87 ) -> None:
88     self.queueError(
89         self.targetAddr,
90         flags=flags,
91         major=error.Major.DISCONNECT,
92         minor=error.DisconnectErrorCodes.CLIENT_DISCONNECT,
93         data=data,
94     )
95
96 def connect(self) -> None:
97     try:
98         logger.info(
99             f"{bcolors.WARNING}# Handshake with {self.targetAddr}
100             starting.{bcolors.ENDC}"
101         )
102         self.outboundThread.start()
103         self.queueAuth(self.targetAddr, self.cert,
104                         self.ecKey.public_key())
105         authPacket = None

```

```

103     ackPacket = None
104     while True:
105         p, addr = self.receivePacket()
106         if p is not None:
107             # logic
108             if p.packet_type == packet.Type.AUTH:
109                 logger.info(
110                     f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC}
111                        {bcolors.OKCYAN}{p}{bcolors.ENDC}"
112                 )
113                 authPacket = p
114                 self.sessionKey = auth.generateSessionKey(
115                     self.ecKey, p.public_key
116                 )
117                 if not self.validateCertificate(p.certificate):
118                     logger.critical(f"Invalid peer cert
119                        {p.certificate}")
120                     self.queueError(
121                         major=error.Major.CONNECTION,
122                         minor=error.ConnectionErrorCodes.CERTIFICATE_INVALID,
123                         data=b"Invalid Certificate.",
124                     )
125                     break
126                 self.queueFinished(
127                     self.targetAddr, p.sequence_id, self.sessionKey
128                 )
129             elif p.packet_type == packet.Type.ACK:
130                 logger.info(
131                     f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC}
132                        {bcolors.OKCYAN}{p}{bcolors.ENDC}"
133                 )
134                 ackPacket = p
135                 self.receiveAck(p, addr)
136             elif p.packet_type == packet.Type.ERROR:
137                 logger.info(
138                     f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC}
139                        {bcolors.OKCYAN}{p}{bcolors.ENDC}"
140                 )
141                 self.receive(p, addr)
142             else:
143                 logger.warning(
144                     f"{bcolors.WARNING}! {addr} :{bcolors.ENDC}
145                        {bcolors.WARNING}{p}{bcolors.ENDC}"
146                 )
147             if authPacket is not None and ackPacket is not None:

```

```

143         break
144     else:
145         # timeout and abort
146         logger.critical("Server not responsive.")
147     if self.validateHandshake(p.data):
148         # success
149         logger.info(
150             f"{bcolors.OKGREEN}Handshake success starting
151               mainloop...{bcolors.ENDC}"
152         )
153         self.inboundThread.start()
154         if self.onConnect:
155             self.onConnect(addr)
156     else:
157         logger.critical(
158             f"Local finished value
159               {node.Node._generateFinished(self.sessionKey)} does
160               not match peer finished value {ackPacket.data}"
161         )
162         self.queueError(
163             major=error.Major.CONNECTION,
164             minor=error.ConnectionErrorCodes.FINISH_INVALID,
165             data=b"Invalid finish.",
166         )
167     except error.PaperClipError as e:
168         raise e
169     except Exception as e:
170         raise e
171     else:
172         if self.isRunning.is_set():
173             self._quit()
174
175 # auth
176 def validateCertificate(self, certificate: auth.x509.Certificate) -> bool:
177     url = f"http://{self.targetHost}:5000/auth/certificate/validate"
178     headers = {"Content-Type": "application/json"}
179     certificate = base64.encodebytes(
180         auth.getDerFromCertificate(certificate)
181     ).decode()
182     data = {"certificate": certificate}
183     r = requests.get(url, headers=headers, data=json.dumps(data))
184     if r.status_code == 200:
185         return r.json()["valid"]
186     else:
187         return False

```

```

185
186 # misc
187 def quit(self, msg: str = "quit call", e: Exception = None) -> None:
188     self.queueDisconnect(data=msg.encode())
189     self.queue.join()
190     super().quit(msg, e)
191
192 def handleDisconnectError(
193     self, p: packet.ErrorPacket, addr: tuple[str, int], e:
194         error.DisconnectError
195 ) -> None:
196     match e:
197         case error.ServerDisconnectError():
198             self._quit(e)
199         case error.ClientDisconnectError():
200             pass # should not react to client disconnect
201         case _:
202             raise e
203
204 def mainloop(self, onQuit=None) -> None:
205     try:
206         while self.isRunning.is_set():
207             pass
208     except KeyboardInterrupt:
209         print(f"{bcolors.FAIL}Quitting. Please wait...{bcolors.ENDC}")
210     finally:
211         if onQuit is None:
212             self.quit(e=self.exitError)
213         else:
214             onQuit(e=self.exitError)

```

```

1 # udp.error
2 from enum import Enum
3
4
5 class Major(Enum):
6     ERROR = 0
7     CONNECTION = 1
8     DISCONNECT = 2
9     PACKET = 3
10
11
12 class Minor(Enum):

```

```

13     pass
14
15
16 class PaperClipError(Exception):
17     """Unknown error"""
18
19
20 # connection
21 class ConnectionErrorCodes(Minor):
22     CONNECTION = 0
23     NO_SPACE = 1
24     CERTIFICATE_INVALID = 2
25     FINISH_INVALID = 3
26
27
28 class ConnectionError(PaperClipError):
29     """Handshake connection could not be finished"""
30
31
32 class NoSpaceError(ConnectionError):
33     """Server has insufficient space to accept new clients"""
34
35
36 class CertificateInvalidError(ConnectionError):
37     """Certificate is invalid / can not be validated"""
38
39
40 class FinishInvalidError(ConnectionError):
41     """Finish is invalid"""
42
43
44 _connectionErrors = {
45     ConnectionErrorCodes.CONNECTION: ConnectionError,
46     ConnectionErrorCodes.NO_SPACE: NoSpaceError,
47     ConnectionErrorCodes.CERTIFICATE_INVALID: CertificateInvalidError,
48     ConnectionErrorCodes.FINISH_INVALID: FinishInvalidError,
49 }
50
51
52 def getConnectionError(minor: ConnectionErrorCodes | int) -> ConnectionError:
53     try:
54         minor = minor if isinstance(minor, Minor) else
55             ConnectionErrorCodes(minor)
56         if minor in _connectionErrors:
57             return _connectionErrors[minor]

```



```

57         else:
58             return PaperClipError
59     except ValueError:
60         return PaperClipError
61
62
63 def getConnectionCode(error: ConnectionError) -> ConnectionErrorCodes:
64     try:
65         return list(_connectionErrors.keys())[
66             list(_connectionErrors.values()).index(error)
67         ]
68     except ValueError:
69         return PaperClipError
70
71
72 # disconnect
73 class DisconnectErrorCodes(Minor):
74     DISCONNECT = 0
75     SERVER_DISCONNECT = 1
76     CLIENT_DISCONNECT = 2
77
78
79 class DisconnectError(PaperClipError):
80     """A party is disconnecting"""
81
82
83 class ServerDisconnectError(DisconnectError):
84     """The server is closing"""
85
86
87 class ClientDisconnectError(DisconnectError):
88     """The client is closing"""
89
90
91 _disconnectErrors = {
92     DisconnectErrorCodes.DISCONNECT: DisconnectError,
93     DisconnectErrorCodes.SERVER_DISCONNECT: ServerDisconnectError,
94     DisconnectErrorCodes.CLIENT_DISCONNECT: ClientDisconnectError,
95 }
96
97
98 def getDisconnectError(minor: DisconnectErrorCodes | int) -> DisconnectError:
99     try:
100         minor = minor if isinstance(minor, Minor) else
            DisconnectErrorCodes(minor)

```

```

101         if minor in _disconnectErrors:
102             return _disconnectErrors[minor]
103         else:
104             return PaperClipError
105     except ValueError:
106         return PaperClipError
107
108
109 def getDisconnectCode(error: DisconnectError) -> DisconnectErrorCodes:
110     try:
111         return list(_disconnectErrors.keys())[
112             list(_disconnectErrors.values()).index(error)
113         ]
114     except ValueError:
115         return PaperClipError
116
117
118 # packet
119 class PacketErrorCodes(Minor):
120     PACKET = 0
121     VERSION = 1
122     PACKET_TYPE = 2
123     FLAGS = 3
124     SEQUENCE_ID = 4
125     FRAGMENT_ID = 5
126     FRAGMENT_NUMBER = 6
127     INIT_VECTOR = 7
128     COMPRESSION = 8
129     CHECKSUM = 9
130
131
132 class PacketError(PaperClipError):
133     """Packet cannot be read"""
134
135
136 class VersionError(PacketError):
137     """Packet Version is invalid / does not match expected"""
138
139
140 class PacketTypeError(PacketError):
141     """Packet Type is invalid / unknown"""
142
143
144 class FlagsError(PacketError):
145     """Flags are invalid / unknown"""

```

```

146
147
148 class SequenceIdError(PacketError):
149     """Sequence Id is invalid / does not match expected"""
150
151
152 class FragmentIdError(PacketError):
153     """Fragment Id is invalid / unknown"""
154
155
156 class FragmentNumberError(PacketError):
157     """Fragment Number is invalid / unknown"""
158
159
160 class InitVectorError(PacketError):
161     """Init Vector is invalid / unknown i.e. decrypt fail"""
162
163
164 class CompressionError(PacketError):
165     """Decompression fail"""
166
167
168 class ChecksumError(PacketError):
169     """Checksum is invalid / unknown i.e. checksum fail"""
170
171
172 _packetErrors = {
173     PacketErrorCodes.PACKET: PacketError,
174     PacketErrorCodes.VERSION: VersionError,
175     PacketErrorCodes.PACKET_TYPE: PacketTypeError,
176     PacketErrorCodes.FLAGS: FlagsError,
177     PacketErrorCodes.SEQUENCE_ID: SequenceIdError,
178     PacketErrorCodes.FRAGMENT_ID: FragmentIdError,
179     PacketErrorCodes.FRAGMENT_NUMBER: FragmentNumberError,
180     PacketErrorCodes.INIT_VECTOR: InitVectorError,
181     PacketErrorCodes.COMPRESSION: CompressionError,
182     PacketErrorCodes.CHECKSUM: ChecksumError,
183 }
184
185
186 def getPacketError(minor: PacketErrorCodes | int) -> PacketError:
187     try:
188         minor = minor if isinstance(minor, Minor) else PacketErrorCodes(minor)
189         if minor in _packetErrors:
190             return _packetErrors[minor]

```

```

191         else:
192             return PaperClipError
193     except ValueError:
194         return PaperClipError
195
196
197 def getPacketCode(error: PacketError) -> PacketErrorCodes:
198     try:
199         return
200         list(_packetErrors.keys())[list(_packetErrors.values()).index(error)]
201     except ValueError:
202         return PaperClipError
203
204 # convenience
205 def getError(major: Major | int, minor: Minor | int = 0) -> PaperClipError:
206     try:
207         major = major if isinstance(major, Major) else Major(major)
208         match major:
209             case Major.CONNECTION:
210                 return getConnectionError(minor)
211             case Major.DISCONNECT:
212                 return getDisconnectError(minor)
213             case Major.PACKET:
214                 return getPacketError(minor)
215             case _:
216                 return PaperClipError
217     except TypeError:
218         return PaperClipError
219
220
221 def getMinor(major: Major, minor: int) -> Minor:
222     match major:
223         case Major.CONNECTION:
224             return ConnectionErrorCodes(minor)
225         case Major.DISCONNECT:
226             return DisconnectErrorCodes(minor)
227         case Major.PACKET:
228             return PacketErrorCodes(minor)
229         case _:
230             return Minor
231
232
233 def getErrorCode(error: PaperClipError) -> tuple[Major, Minor]:
234     match error:

```

```

235         case c if issubclass(c, ConnectionError):
236             return (Major.CONNECTION, getConnectionCode(error))
237         case d if issubclass(d, DisconnectError):
238             return (Major.DISCONNECT, getDisconnectCode(error))
239         case p if issubclass(p, PacketError):
240             return (Major.PACKET, getPacketCode(error))
241         case _:
242             return (Major.ERROR, Minor)

```

```

1  # udp.node
2  import time
3  from datetime import datetime
4  from queue import Empty, Queue
5  from socket import SOCK_DGRAM
6  from socket import socket as Socket
7  from threading import Event, Lock, Thread, get_ident
8
9  from cryptography.hazmat.primitives.asymmetric.ec import
    EllipticCurvePrivateKey
10 from cryptography.x509 import Certificate
11
12 from . import (
13     QUEUE_TIMEOUT,
14     SEND_SLEEP_TIME,
15     SOCKET_BUFFER_SIZE,
16     SOCKET_TIMEOUT,
17     auth,
18     bcolors,
19     error,
20     logger,
21     packet,
22 )
23
24 ACK_RESET_SIZE = (2**packet.ACK_BITS_SIZE) // 2
25
26
27 class Node:
28     addr: tuple[str, int]
29     _sequenceId: int
30     sentAckBits = list[bool | None]
31     recvAckBits = list[bool | None]
32     newestSeqId: int | None
33     fragBuffer: dict[int, list[packet.Packet]]

```

```

34 queue: Queue
35 heartbeat: datetime | None
36 # id
37 cert: Certificate | None
38 _accountId: int | None
39 # session
40 ecKey: EllipticCurvePrivateKey
41 sessionKey: bytes | None
42 handshake: bool
43 # threads
44 inboundThread: Thread
45 outboundThread: Thread
46 sequenceIdLock: Lock
47 sendLock: Lock
48 isRunning: Event
49 # socket
50 socket: Socket | None
51 # callback
52 onReceiveData: None
53 # exitCode
54 exitError: error.PaperClipError | None
55
56 def __init__(
57     self,
58     addr: tuple[str, int],
59     cert: Certificate | None = None,
60     accountId: int | None = None,
61     sendLock: Lock = Lock(),
62     socket: Socket | None = Socket(type=SOCK_DGRAM),
63     onReceiveData: None = None,
64 ) -> None:
65     self.addr = addr
66     self.sequenceId = 0
67     self.sentAckBits = [None for _ in range(2**packet.ACK_BITS_SIZE)]
68     self.recvAckBits = [None for _ in range(2**packet.ACK_BITS_SIZE)]
69     self.newestSeqId = 0
70     self.fragBuffer = {}
71     self.queue = Queue()
72     # id
73     self.cert = cert
74     self.accountId = accountId
75     # session
76     self.sessionKey = None
77     self.handshake = False
78     # threads

```

```

79     self.inboundThread = Thread(
80         name=f"{self.port}:Inbound", target=self.listen, daemon=True
81     )
82     self.outboundThread = Thread(
83         name=f"{self.port}:Outbound", target=self.sendQueue, daemon=True
84     )
85     self.sequenceIdLock = Lock()
86     self.sendLock = sendLock
87     self.isRunning = Event()
88     self.isRunning.set()
89     # socket
90     self.socket = socket
91     self.socket.settimeout(SOCKET_TIMEOUT)
92     # callback
93     self.onReceiveData = onReceiveData
94     # exit
95     self.exitError = None
96
97     def bind(self, addr):
98         self.socket.bind(addr)
99
100     # properties
101     @property
102     def host(self) -> str:
103         return self.addr[0]
104
105     @property
106     def port(self) -> int:
107         return self.addr[1]
108
109     @property
110     def sequenceId(self) -> int:
111         return self._sequenceId
112
113     @sequenceId.setter
114     def sequenceId(self, v: int) -> None:
115         self._sequenceId = v % 2**packet.SEQUENCE_ID_SIZE
116
117     def incrementSequenceId(self, addr: tuple[str, int]) -> None:
118         with self.getSequenceIdLock(addr):
119             self.sequenceId += 1
120
121     @property
122     def accountId(self) -> int:
123         return self._accountId

```

```

124
125 @accountId.setter
126 def accountId(self, v: int | str | None) -> None:
127     try:
128         self._accountId = int(v)
129     except ValueError:
130         self._accountId = v
131     except TypeError:
132         self._accountId = None
133
134 def getSentAckBit(self, addr: tuple[str, int], p: packet.Packet) -> bool
    | None:
135     return self.sentAckBits[p.sequence_id]
136
137 def setSentAckBit(self, addr: tuple[str, int], ackBit: int, v: bool) ->
    None:
138     self.sentAckBits[ackBit] = v
139
140 def getSentAckBits(self, addr: tuple[str, int]) -> list[bool | None]:
141     return self.sentAckBits
142
143 def getRecvAckBit(self, addr: tuple[str, int], p: packet.Packet) -> bool
    | None:
144     return self.recvAckBits[p.sequence_id]
145
146 def getRecvAckBits(self, addr: tuple[str, int]) -> list[bool | None]:
147     return self.recvAckBits
148
149 def setRecvAckBit(self, addr: tuple[str, int], ackBit: int, v: bool) ->
    None:
150     self.recvAckBits[ackBit] = v
151
152 def resetRecvAckBits(self, addr: tuple[str, int]) -> None:
153     recvAckBits = self.getRecvAckBits(addr)
154     newestSeqId = self.getNewestSeqId(addr)
155     pointer = (newestSeqId - ACK_RESET_SIZE) % 2**packet.ACK_BITS_SIZE
156     counter = 0
157     while counter != pointer:
158         recvAckBits[(newestSeqId + 1 + counter) %
            2**packet.ACK_BITS_SIZE] = None
159         counter += 1
160
161 def getNewestSeqId(self, addr: tuple[str, int]) -> int:
162     return self.newestSeqId
163

```



```

164 def setNewestSeqId(self, addr: tuple[str, int], newestSeqId: int) -> None:
165     self.newestSeqId = newestSeqId
166
167 @staticmethod
168 def getNewerSeqId(currentSeqId: int, newSeqId: int) -> int:
169     currentDiff = (newSeqId - currentSeqId) % (2**packet.SEQUENCE_ID_SIZE)
170     newDiff = (currentSeqId - newSeqId) % (2**packet.SEQUENCE_ID_SIZE)
171     if newDiff < currentDiff:
172         return currentSeqId
173     else:
174         return newSeqId
175
176 def getSessionKey(self, addr: tuple[str, int]) -> int:
177     return self.sessionKey
178
179 def getHandshake(self, addr: tuple[str, int]) -> bool:
180     return self.handshake
181
182 def getFragBuffer(self, addr: tuple[str, int]) -> dict[int,
183     list[packet.Packet]]:
184     return self.fragBuffer
185
186 def getSequenceId(self, addr: tuple[str, int]) -> int:
187     return self.sequenceId
188
189 def getSequenceIdLock(self, addr: tuple[str, int]) -> Lock:
190     return self.sequenceIdLock
191
192 def getQueue(self, addr: tuple[str, int]) -> Queue:
193     return self.queue
194
195 def getHeartbeat(self, addr: tuple[str, int]) -> datetime:
196     return self.heartbeat
197
198 def setHeartbeat(self, addr: tuple[str, int], v: datetime) -> None:
199     self.heartbeat = v
200
201 def regenerateEcKey(self) -> None:
202     self.ecKey = auth.generateEcKey()
203
204 # sends
205 def sendPacket(self, addr: tuple[str, int], p: packet.Packet) -> None:
206     with self.sendLock:
207         try:
208             self.socket.sendto(p.pack(p), (addr[0], addr[1]))

```

```

208         logger.info(
209             f"{bcolors.OKBLUE}> {addr} :{bcolors.ENDC}"
210             f"{bcolors.OKCYAN}{p}{bcolors.ENDC}"
211         )
212     except error.PacketError as e:
213         logger.error(
214             f"{bcolors.FAIL}# > {bcolors.ENDC}{bcolors.OKBLUE}{addr}"
215             f":{bcolors.ENDC}"
216             f"{bcolors.FAIL}{type(e).__name__}:{e.args[0] if"
217             f"len(e.args) > 0 else ''}{p}{bcolors.ENDC}"
218         )
219
220 def sendQueue(self) -> None:
221     while self.isRunning.is_set():
222         try:
223             addr, p = self.queue.get(timeout=QUEUE_TIMEOUT)
224             if p.flags[packet.Flag.RELIABLE.value]:
225                 if self.getSentAckBit(addr, p):
226                     self.queue.task_done()
227                     continue
228                 else:
229                     self.sendPacket(addr, p)
230                     self.queue.task_done()
231                     self.queue.put((addr, p))
232             else:
233                 self.sendPacket(addr, p)
234                 self.queue.task_done()
235                 time.sleep(SEND_SLEEP_TIME)
236         except Empty:
237             pass # check still running
238     else:
239         logger.info("| sendQueue thread stopping...")
240
241 def queuePacket(self, addr: tuple[str, int], p: packet.Packet) -> None:
242     if p.flags[packet.Flag.RELIABLE.value]:
243         self.setSentAckBit(addr, p.sequence_id, False)
244     if p.flags[packet.Flag.CHECKSUM.value]:
245         p.setChecksum()
246     if p.flags[packet.Flag.COMPRESSED.value]:
247         p.compressData()
248     if p.flags[packet.Flag.ENCRYPTED.value]:
249         p.encryptData(self.getSessionKey(addr))
250     if p.flags[packet.Flag.FRAG.value]:
251         frags = p.fragment()
252         for frag in frags:

```

```

249         self.getQueue(addr).put((addr, frag))
250     else:
251         self.getQueue(addr).put((addr, p))
252
253     def queueDefault(
254         self,
255         addr: tuple[str, int],
256         flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
257         data: bytes | None = None,
258     ) -> None:
259         p = packet.Packet(sequence_id=self.getSequenceId(addr), flags=flags,
260                             data=data)
261         self.incrementSequenceId(addr)
262         self.queuePacket(addr, p)
263
264     def queueACK(
265         self,
266         addr: tuple[str, int],
267         ackId: int,
268         flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
269         data: bytes | None = None,
270     ) -> None:
271         ack_bits = self.packRecvAckBits(self.getRecvAckBits(addr), ackId)
272         p = packet.AckPacket(
273             sequence_id=self.getSequenceId(addr),
274             flags=flags,
275             ack_id=ackId,
276             ack_bits=ack_bits,
277             data=data,
278         )
279         self.incrementSequenceId(addr)
280         self.queuePacket(addr, p)
281
282     def queueAuth(
283         self,
284         addr: tuple[str, int],
285         cert: Certificate,
286         publicEc: auth.ec.EllipticCurvePublicKey,
287     ) -> None:
288         p = packet.AuthPacket(
289             sequence_id=self.getSequenceId(addr), certificate=cert,
290             public_key=publicEc
291         )
292         self.incrementSequenceId(addr)
293         self.queuePacket(addr, p)

```

```

292
293 def queueFinished(
294     self, addr: tuple[str, int], seqId: int, sessionKey: bytes
295 ) -> None:
296     finished = Node._generateFinished(sessionKey)
297     self.queueACK(addr, seqId, data=finished)
298
299 @staticmethod
300 def _generateFinished(sessionKey: bytes) -> bytes:
301     return auth.generateFinished(
302         sessionKey, finishedLabel=b"node finished", messages=b"\x13"
303     )
304
305 def queueHeartbeat(
306     self,
307     addr: tuple[str, int],
308     heartbeat: bool,
309     flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
310     data: bytes | None = None,
311 ) -> None:
312     p = packet.HeartbeatPacket(
313         sequence_id=self.getSequenceId(addr),
314         flags=flags,
315         heartbeat=heartbeat,
316         data=data,
317     )
318     self.incrementSequenceId(addr)
319     self.queuePacket(addr, p)
320
321 def queueError(
322     self,
323     addr: tuple[str, int],
324     major: error.Major | int,
325     minor: error.Minor | int,
326     flags: list[int] = [0 for _ in range(packet.FLAGS_SIZE)],
327     data: bytes | None = None,
328 ) -> None:
329     sId = self.getSequenceId(addr)
330     p = packet.ErrorPacket(
331         sequence_id=sId if sId is not None else 0,
332         flags=flags,
333         major=major,
334         minor=minor,
335         data=data,
336     )

```

```

337         if sId is not None:
338             self.incrementSequenceId(addr)
339         self.queuePacket(addr, p)
340
341     def queueDisconnect(
342         self,
343         addr: tuple[str, int],
344         flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
345         data: bytes | None = None,
346     ) -> None:
347         self.queueError(
348             addr,
349             flags=flags,
350             major=error.Major.DISCONNECT,
351             minor=error.DisconnectErrorCodes.DISCONNECT,
352             data=data,
353         )
354
355     # receives
356     def receivePacket(
357         self,
358     ) -> tuple[packet.Packet, tuple[str, int]] | tuple[None, None]:
359         try:
360             data, addr = self.socket.recvfrom(SOCKET_BUFFER_SIZE)
361             try:
362                 p = packet.unpack(data)
363                 return p, addr
364             except error.PacketError as e:
365                 logger.error(
366                     f"{bcolors.FAIL}# < {bcolors.ENDC}{bcolors.OKBLUE}-{addr}
367                     :{bcolors.ENDC}
368                     {bcolors.FAIL}{type(e).__name__}:{e.args[0] if
369                     len(e.args) > 0 else ''}{p}{bcolors.ENDC}"
370                 )
371                 major, minor = error.getErrorCod(e)
372                 self.queueError(addr, major, minor)
373                 return None, None
374             except ConnectionResetError:
375                 return None, None
376             except TimeoutError:
377                 return None, None
378
379     def receive(
380         self, p: packet.Packet, addr: tuple[str, int]
381     ) -> tuple[packet.Packet, tuple[str, int]] | None:

```

```

379     if p is not None:
380         if self.handleFlags(p, addr):
381             match p.packet_type:
382                 case packet.Type.DEFAULT:
383                     logger.info(
384                         f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC}
385                             {bcolors.OKCYAN}{p}{bcolors.ENDC}"
386                     )
387                     return self.receiveDefault(p, addr)
388                 case packet.Type.ACK:
389                     logger.info(
390                         f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC}
391                             {bcolors.OKCYAN}{p}{bcolors.ENDC}"
392                     )
393                     return self.receiveAck(p, addr)
394                 case packet.Type.AUTH:
395                     logger.info(
396                         f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC}
397                             {bcolors.OKCYAN}{p}{bcolors.ENDC}"
398                     )
399                     return self.receiveAuth(p, addr)
400                 case packet.Type.HEARTBEAT:
401                     logger.info(
402                         f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC}
403                             {bcolors.OKCYAN}{p}{bcolors.ENDC}"
404                     )
405                     return self.receiveHeartbeat(p, addr)
406                 case packet.Type.ERROR:
407                     logger.warning(
408                         f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC}
409                             {bcolors.FAIL}{p}{bcolors.ENDC}"
410                     )
411                     try:
412                         return self.receiveError(p, addr)
413                     except error.PaperClipError as e:
414                         self.handleError(p, addr, e)
415                 case _:
416                     logger.warning(
417                         f"Unknown packet type '{p.packet_type}' for
418                             packet {p}"
419                     )
420                     self.queueError(
421                         addr,
422                         major=error.Major.PACKET,
423                         minor=error.PacketErrorCodes.PACKET_TYPE,

```

```

418             data=p.sequence_id,
419         )
420
421     def receiveDefault(
422         self, p: packet.Packet, addr: tuple[str, int]
423     ) -> tuple[packet.Packet, tuple[str, int]]:
424         self.setNewestSeqId(
425             addr, self.getNewerSeqId(self.getNewestSeqId(addr), p.sequence_id)
426         )
427         if self.onReceiveData:
428             self.onReceiveData(addr, p.data)
429         return (p, addr)
430
431     def receiveAck(
432         self, p: packet.AckPacket, addr: tuple[str, int]
433     ) -> tuple[packet.Packet, tuple[str, int]]:
434         self.setNewestSeqId(
435             addr, self.getNewerSeqId(self.getNewestSeqId(addr), p.sequence_id)
436         )
437         self.setSentAckBit(addr, p.ack_id, True)
438         # set all bits from ack bits to true (to mitigate lost ack)
439         for i, j in enumerate(
440             range(p.ack_id - 1, p.ack_id - 1 - packet.ACK_BITS_SIZE, -1)
441         ):
442             if p.ack_bits[i]:
443                 self.setSentAckBit(addr, j, True)
444         return (p, addr)
445
446     def receiveAuth(
447         self, p: packet.AuthPacket, addr: tuple[str, int]
448     ) -> tuple[packet.Packet, tuple[str, int]]:
449         raise NotImplementedError(
450             "Node should not receive auth. A child class must overwrite."
451         )
452         return (p, addr)
453
454     def receiveHeartbeat(
455         self, p: packet.HeartbeatPacket, addr: tuple[str, int]
456     ) -> tuple[packet.Packet, tuple[str, int]]:
457         if not p.heartbeat:
458             self.queueHeartbeat(addr, heartbeat=True)
459             pass
460         return (p, addr)
461
462     def receiveError(self, p: packet.ErrorPacket, addr: tuple[str, int]) ->

```

```

None:
463     raise error.getError(p.major, p.minor)(p.data)
464
465 def listen(self) -> None:
466     logger.info(
467         f"{bcolors.HEADER}Listening @
           {self.socket.getsockname()}{bcolors.ENDC}"
468     )
469     while self.isRunning.is_set():
470         p, addr = self.receivePacket()
471         self.receive(p, addr)
472     else:
473         logger.info("| listen thread stopping...")
474
475 # flags handle
476 def handleFlags(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
477     # defrag -> decrypt -> decompress -> validate checksum -> reliable
478     if self.handleFrag(p, addr):
479         return False
480     else:
481         self.handleEncrypted(p, addr)
482         self.handleCompressed(p, addr)
483         self.handleChecksum(p, addr)
484         self.handleReliable(p, addr)
485         return True
486
487 def handleReliable(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
488     if p.flags[packet.Flag.RELIABLE.value]:
489         self.setNewestSeqId(
490             addr, self.getNewerSeqId(self.getNewestSeqId(addr),
491                                     p.sequence_id)
492         )
493         self.setRecvAckBit(addr, p.sequence_id, True)
494         self.resetRecvAckBits(addr)
495         self.queueACK(addr, p.sequence_id)
496         return True
497     else:
498         return False
499
500 def handleFrag(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
501     if p.flags[packet.Flag.FRAG.value]:
502         logger.info(
503             f"\t{bcolors.OKBLUE}< {addr} :{bcolors.ENDC}{bcolors.WARNING}
              FRAG {p.fragment_id}/{p.fragment_number} {p}{bcolors.ENDC}"
504         )

```



```

504         if p.sequence_id not in self.getFragBuffer(addr):
505             self.getFragBuffer(addr)[p.sequence_id] = [
506                 None for _ in range(p.fragment_number)
507             ]
508         self.getFragBuffer(addr)[p.sequence_id][p.fragment_id] = p
509         if all(self.getFragBuffer(addr)[p.sequence_id]):
510             defrag = p.defragment(self.getFragBuffer(addr)[p.sequence_id])
511             del self.getFragBuffer(addr)[p.sequence_id]
512             self.receive(defrag, addr)
513         return True
514     else:
515         return False
516
517     def handleCompressed(self, p: packet.Packet, addr: tuple[str, int]) ->
518         bool:
519         if p.flags[packet.Flag.COMPRESSED.value]:
520             p.decompressData()
521             return True
522         else:
523             return False
524
525     def handleEncrypted(self, p: packet.Packet, addr: tuple[str, int]) ->
526         bool:
527         if p.flags[packet.Flag.ENCRYPTED.value]:
528             p.decryptData(self.getSessionKey(addr))
529             return True
530         else:
531             return False
532
533     def handleChecksum(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
534         if p.flags[packet.Flag.CHECKSUM.value]:
535             if not p.validateChecksum():
536                 logger.warning(f"\tInvalid checksum: {p}")
537             else:
538                 logger.info(f"\tValid checksum: {p}")
539             return True
540         else:
541             return False
542
543     # error handle
544     def handleError(
545         self, p: packet.ErrorPacket, addr: tuple[str, int], e:
546         error.PaperClipError
547     ) -> None:
548         match e:

```

```

546         case error.ConnectionError():
547             self.handleConnectionError(p, addr, e)
548         case error.DisconnectError():
549             self.handleDisconnectError(p, addr, e)
550         case error.PacketError():
551             self.handlePacketError(p, addr, e)
552         case _:
553             raise e
554
555     def handleConnectionError(
556         self, p: packet.ErrorPacket, addr: tuple[str, int], e:
557         error.ConnectionError
558     ) -> None:
559         match e:
560             case error.NoSpaceError():
561                 return self.quit("no server space", e)
562             case error.CertificateInvalidError():
563                 return self.quit("invalid certificate", e)
564             case error.FinishInvalidError():
565                 return self.quit("invalid finish", e)
566             case _:
567                 raise e
568
569     def handleDisconnectError(
570         self, p: packet.ErrorPacket, addr: tuple[str, int], e:
571         error.DisconnectError
572     ) -> None:
573         match e:
574             case error.ServerDisconnectError:
575                 pass # overwrite
576             case error.ClientDisconnectError:
577                 pass # overwrite
578             case _:
579                 raise e
580
581     def handlePacketError(
582         self, p: packet.ErrorPacket, addr: tuple[str, int], e:
583         error.PacketError
584     ) -> None:
585         match e:
586             case error.VersionError():
587                 pass
588             case error.PacketTypeError():
589                 pass
590             case error.FlagsError():

```

```

588         pass
589     case error.SequenceIdError():
590         pass
591     case error.FragmentIdError():
592         pass
593     case error.FragmentNumberError():
594         pass
595     case error.InitVectorError():
596         pass
597     case error.CompressionError():
598         pass
599     case error.ChecksumError():
600         pass
601     case _:
602         raise e
603
604     # util
605     @staticmethod
606     def packRecvAckBits(recvAckBits: list[bool], ackId: int) -> list[bool |
        None]:
607         return [
608             recvAckBits[i % 2**packet.ACK_BITS_SIZE]
609             for i in range(ackId - 1, ackId - 1 - packet.ACK_BITS_SIZE, -1)
610         ]
611
612     # misc
613     def startThreads(self) -> None:
614         self.inboundThread.start()
615         self.outboundThread.start()
616
617     def validateCertificate(self, certificate: Certificate) -> bool:
618         # overwrite
619         return True
620
621     def validateHandshake(self, finished: bytes) -> bool:
622         self.handshake = Node._generateFinished(self.sessionKey) == finished
623         return self.handshake
624
625     def quit(self, msg: str = "quit call", e: Exception = None) -> None:
626         logMsg = f"{bcolors.FAIL}# Quitting due to {msg}.{bcolors.ENDC}"
627         if e is not None:
628             logger.critical(logMsg)
629         else:
630             logger.info(logMsg)
631         self.isRunning.clear()

```

```

632         if self.inboundThread.is_alive() and get_ident() !=
           self.inboundThread.ident:
633             self.inboundThread.join()
634         if self.outboundThread.is_alive() and get_ident() !=
           self.outboundThread.ident:
635             self.outboundThread.join()
636         self.socket.close()
637         logger.info(f"{bcolors.FAIL}# Quit finished.{bcolors.ENDC}")
638         if e is not None:
639             self.exitError = e
640             if get_ident() in (self.inboundThread.ident,
                               self.outboundThread.ident):
641                 pass
642             else:
643                 raise e
644
645     def _quit(self, e: Exception = None) -> None:
646         self.exitError = e
647         self.isRunning.clear()

```

```

1  # udp.packet
2  import struct
3  from enum import Enum
4
5  from cryptography.hazmat.primitives.asymmetric.ec import
     EllipticCurvePublicKey
6  from cryptography.x509 import Certificate
7
8  from . import auth, error, logger, utils
9
10 VERSION = 0
11 # SIZE in Bits
12 VERSION_SIZE = 4
13 PACKET_TYPE_SIZE = 4
14 FLAGS_SIZE = 8
15 SEQUENCE_ID_SIZE = 16
16 FRAGMENT_ID_SIZE = 8
17 FRAGMENT_NUM_SIZE = 8
18 INIT_VECTOR_SIZE = 16
19 CHECKSUM_SIZE = 16
20 ACK_ID_SIZE = SEQUENCE_ID_SIZE # 16
21 ACK_BITS_SIZE = SEQUENCE_ID_SIZE # 16
22

```

```

23
24 class Type(Enum):
25     DEFAULT = 0
26     ACK = 1
27     AUTH = 2
28     HEARTBEAT = 3
29     ERROR = 4
30
31
32 class Flag(Enum):
33     RELIABLE = 0
34     CHECKSUM = 1
35     COMPRESSED = 2
36     ENCRYPTED = 3
37     FRAG = 4
38
39
40 class Heartbeat(Enum):
41     PING = 0
42     PONG = 1
43
44
45 def lazyFlags(*fs: list[Flag]) -> list[int]:
46     flags = [0 for _ in range(FLAGS_SIZE)]
47     for flag in fs:
48         flags[flag.value] = 1
49     return flags
50
51
52 class Packet:
53     version: int = VERSION
54     packet_type: Type = Type.DEFAULT
55     flags: list[int] = [0 for _ in range(FLAGS_SIZE)]
56     sequence_id: int = 0
57     fragment_id: int | None = None
58     fragment_number: int | None = None
59     init_vector: int | None = None
60     checksum: int | None = None
61     _data: bytes | None = None
62
63     def __init__(
64         self,
65         version: int = VERSION,
66         packet_type: Type = Type.DEFAULT,
67         flags: list[int] = [0 for _ in range(FLAGS_SIZE)],

```

```

68     sequence_id: int = None,
69     fragment_id: int | None = None,
70     fragment_number: int | None = None,
71     init_vector: int | None = None,
72     checksum: int | None = None,
73     data: bytes | None = None,
74 ) -> None:
75     self.version = version
76     self.packet_type = packet_type
77     self.flags = flags
78     self.sequence_id = sequence_id
79     self.fragment_id = fragment_id
80     self.fragment_number = fragment_number
81     self.init_vector = init_vector
82     self.checksum = checksum
83     self.data = data
84
85 # util
86 def encryptData(self, session_key: bytes) -> None:
87     try:
88         self.flags[Flag.ENCRYPTED.value] = 1
89         iv = (
90             self.init_vector
91             if self.init_vector is not None
92             else auth.generateInitVector()
93         )
94         cipher, iv = auth.generateCipher(session_key, iv)
95         self.init_vector = iv
96         self.data = auth.encryptBytes(cipher, self.data)
97     except Exception as e:
98         raise error.InitVectorError(e)
99
100 def decryptData(self, session_key: bytes) -> None:
101     try:
102         if self.flags[Flag.ENCRYPTED.value]:
103             cipher = auth.generateCipher(session_key, self.init_vector)[0]
104             self.data = auth.decryptBytes(cipher, self.data)
105         else:
106             logger.warning(
107                 f"Packet {self} is not flagged as ENCRYPTED
108                     ({self.flags})."
109             )
110     except Exception as e:
111         raise error.InitVectorError(e)

```

```

112 def compressData(self) -> None:
113     try:
114         self.flags[Flag.COMPRESSED.value] = 1
115         self.data = utils.compressData(self.data)
116     except Exception as e:
117         raise error.CompressionError(e)
118
119 def decompressData(self) -> None:
120     try:
121         if self.flags[Flag.COMPRESSED.value]:
122             self.data = utils.decompressData(self.data)
123         else:
124             logger.warning(
125                 f"Packet {self} is not flagged as COMPRESSED
126                     ({self.flags})."
127             )
128     except Exception as e:
129         raise error.CompressionError(e)
130
131 def setChecksum(self) -> None:
132     try:
133         self.flags[Flag.CHECKSUM.value] = 1
134         data = self.data if self.data is not None else b""
135         self.checksum = utils.generateChecksum(data)
136     except Exception as e:
137         raise error.ChecksumError(e)
138
139 def validateChecksum(self) -> bool:
140     try:
141         if self.flags[Flag.CHECKSUM.value]:
142             data = self.data if self.data is not None else b""
143             return self.checksum == utils.generateChecksum(data)
144         else:
145             logger.warning(
146                 f"Packet {self} is not flagged as CHECKSUM
147                     ({self.flags})."
148             )
149     except Exception as e:
150         raise error.ChecksumError(e)
151
152 @staticmethod
153 def _getHeader(p) -> dict:
154     header = {
155         k: v
156         for k, v in vars(p).items()
157     }

```

```

155         if k not in ("data", "fragment_id", "fragment_number")
156     }
157     return header
158
159 def fragment(self):
160     self.flags[Flag.FRAG.value] = 1
161     header = Packet._getHeader(self)
162     fragData = utils.fragmentData(self.data)
163     fragment_number = len(fragData)
164     return [
165         self._createFragment(
166             header, fragment_id=i, fragment_number=fragment_number,
167             data=data
168         )
169         for i, data in enumerate(fragData)
170     ]
171
172 @classmethod
173 def _createFragment(
174     cls, header: dict, fragment_id: int, fragment_number: int, data: bytes
175 ):
176     return cls(
177         **header,
178         fragment_id=fragment_id,
179         fragment_number=fragment_number,
180         data=data,
181     )
182
183 @classmethod
184 def defragment(cls, frags):
185     if frags[0].flags[Flag.FRAG.value]:
186         header = Packet._getHeader(frags[0])
187         header["flags"][Flag.FRAG.value] = 0
188         data = utils.defragmentData([frag.data for frag in frags])
189         return cls(**header, data=data)
190     else:
191         logger.warning(
192             f"Packet {frags[0]} is not flagged as FRAG
193             ({frags[0].flags})."
194         )
195
196 # dunder
197 def __str__(self) -> str:
198     try:
199         s = self.pack(self)

```



```

198     except error.PaperClipError:
199         s = b""
200         data = self.data if self.data is not None else b""
201         pSize = len(s)
202         dSize = len(data)
203         if len(data) > 12:
204             data = f"{data[:11]}...{str(data[-1:])[1:]}"
205         return f"<{self.version}:{self.packet_type.name} {self.sequence_id}
           {''.join(map(str,self.flags))} {data} [{pSize}:{dSize}]>"
206
207     def __eq__(self, other) -> bool:
208         if isinstance(other, self.__class__):
209             return vars(self) == vars(other)
210         else:
211             return False
212
213     # encode / decode
214     @staticmethod
215     def _encodeVersion(version: int) -> int:
216         try:
217             return version
218         except Exception as e:
219             raise error.VersionError(e)
220
221     @staticmethod
222     def _decodeVersion(version: int) -> int:
223         try:
224             return version
225         except Exception as e:
226             raise error.VersionError(e)
227
228     @staticmethod
229     def _encodeType(packet_type: Type) -> int:
230         try:
231             return packet_type.value
232         except Exception as e:
233             raise error.PacketTypeError(e)
234
235     @staticmethod
236     def _decodeType(packet_type: int) -> Type:
237         try:
238             return Type(packet_type)
239         except Exception as e:
240             raise error.PacketTypeError(e)
241

```

```

242 @staticmethod
243 def encodeVersionType(version: int, packet_type: Type) -> bytes:
244     return struct.pack(
245         "!B",
246         (Packet._encodeVersion(version) * 16) |
247         Packet._encodeType(packet_type),
248     )
249
250 @staticmethod
251 def decodeVersionType(versionType: bytes) -> tuple[int, Type]:
252     versionType = struct.unpack("!B", versionType)[0]
253     version = Packet._decodeVersion(versionType >> 4)
254     packet_type = Packet._decodeType(versionType & 15)
255     return version, packet_type
256
257 @staticmethod
258 def encodeFlags(flags: list[int]) -> bytes:
259     try:
260         return struct.pack("!B", int("".join(map(str, flags)), 2))
261     except Exception as e:
262         raise error.FlagsError(e)
263
264 @staticmethod
265 def decodeFlags(flags: bytes) -> list[int]:
266     try:
267         flags = struct.unpack("!B", flags)[0]
268         flags = [(flags >> i) & 1 for i in range(FLAGS_SIZE)]
269         flags.reverse()
270         return flags
271     except Exception as e:
272         raise error.FlagsError(e)
273
274 @staticmethod
275 def encodeSequenceId(sequence_id: int) -> bytes:
276     try:
277         return struct.pack("!I", sequence_id)
278     except Exception as e:
279         raise error.SequenceIdError(e)
280
281 @staticmethod
282 def decodeSequenceId(sequence_id: bytes) -> int:
283     try:
284         return struct.unpack("!I", sequence_id)[0]
285     except Exception as e:
286         raise error.SequenceIdError(e)

```

```

286
287 @staticmethod
288 def encodeFragmentId(fragment_id: int) -> bytes:
289     try:
290         return struct.pack("!B", fragment_id)
291     except Exception as e:
292         raise error.FragmentIdError(e)
293
294 @staticmethod
295 def decodeFragmentId(fragment_id: bytes) -> int:
296     try:
297         return struct.unpack("!B", fragment_id)[0]
298     except Exception as e:
299         raise error.FragmentIdError(e)
300
301 @staticmethod
302 def encodeFragmentNumber(fragment_number: int) -> bytes:
303     try:
304         return struct.pack("!B", fragment_number)
305     except Exception as e:
306         raise error.FragmentNumberError(e)
307
308 @staticmethod
309 def decodeFragmentNumber(fragment_number: bytes) -> int:
310     try:
311         return struct.unpack("!B", fragment_number)[0]
312     except Exception as e:
313         raise error.FragmentNumberError(e)
314
315 @staticmethod
316 def encodeInitVector(init_vector: bytes) -> bytes:
317     try:
318         return init_vector
319     except Exception as e:
320         raise error.InitVectorError(e)
321
322 @staticmethod
323 def decodeInitVector(init_vector: bytes) -> bytes:
324     try:
325         return init_vector
326     except Exception as e:
327         raise error.InitVectorError(e)
328
329 @staticmethod
330 def encodeChecksum(checksum: int) -> bytes:

```

```

331         try:
332             return struct.pack("!I", checksum)
333         except Exception as e:
334             raise error.ChecksumError(e)
335
336     @staticmethod
337     def decodeChecksum(checksum: bytes) -> int:
338         try:
339             return struct.unpack("!I", checksum)[0]
340         except Exception as e:
341             raise error.ChecksumError(e)
342
343     @staticmethod
344     def encodeHeader(
345         version: int,
346         packet_type: Type,
347         flags: list[int],
348         sequence_id: int,
349         fragment_id: int | None = None,
350         fragment_number: int | None = None,
351         init_vector: int | None = None,
352         checksum: int | None = None,
353     ) -> bytes:
354         versionType = Packet.encodeVersionType(version, packet_type)
355         flags = Packet.encodeFlags(flags)
356         sequence_id = Packet.encodeSequenceId(sequence_id)
357         fragment_id = (
358             Packet.encodeFragmentId(fragment_id) if fragment_id is not None
359             else b""
360         )
361         fragment_number = (
362             Packet.encodeFragmentNumber(fragment_number)
363             if fragment_number is not None
364             else b""
365         )
366         init_vector = (
367             Packet.encodeInitVector(init_vector) if init_vector is not None
368             else b""
369         )
370         checksum = Packet.encodeChecksum(checksum) if checksum is not None
371         else b""
372         return (
373             versionType
374             + flags
375             + sequence_id

```

```

373         + fragment_id
374         + fragment_number
375         + init_vector
376         + checksum
377     )
378
379     @staticmethod
380     def decodeHeader(
381         header: bytes,
382     ) -> tuple[
383         int, Type, list[int], int, int | None, int | None, int | None, int |
384         None, int
385     ]:
386         version, packet_type = Packet.decodeVersionType(header[0:1])
387         flags = Packet.decodeFlags(header[1:2])
388         sequence_id = Packet.decodeSequenceId(header[2:6])
389         offset = 6
390         if flags[Flag.FRAG.value]:
391             fragment_id = Packet.decodeFragmentId(header[offset : offset + 1])
392             fragment_number = Packet.decodeFragmentNumber(
393                 header[offset + 1 : offset + 2]
394             )
395             offset += 2
396         else:
397             fragment_id = None
398             fragment_number = None
399         if flags[Flag.ENCRYPTED.value]:
400             init_vector = Packet.decodeInitVector(header[offset : offset +
401                 16])
402             offset += 16
403         else:
404             init_vector = None
405         if flags[Flag.CHECKSUM.value]:
406             checksum = Packet.decodeChecksum(header[offset : offset + 4])
407             offset += 4
408         else:
409             checksum = None
410         return (
411             version,
412             packet_type,
413             flags,
414             sequence_id,
415             fragment_id,
416             fragment_number,
417             init_vector,
418             checksum,
419         )

```

```

416         checksum,
417         offset,
418     )
419
420     # pack / unpack
421     @classmethod
422     def _packHeader(cls, p) -> bytes:
423         header = cls.encodeHeader(
424             p.version,
425             p.packet_type,
426             p.flags,
427             p.sequence_id,
428             p.fragment_id,
429             p.fragment_number,
430             p.init_vector,
431             p.checksum,
432         )
433         return header
434
435     @classmethod
436     def pack(cls, p) -> bytes:
437         header = cls._packHeader(p)
438         data = p.data if p.data is not None else b""
439         return header + data
440
441     @classmethod
442     def _unpackHeader(cls, bytesP: bytes):
443         *header, offset = cls.decodeHeader(bytesP)
444         return *header, offset
445
446     @classmethod
447     def unpack(cls, bytesP: bytes):
448         *header, offset = cls._unpackHeader(bytesP)
449         data = bytesP[offset:] if offset < len(bytesP) else None
450         return cls(*header, data=data)
451
452
453 class AckPacket(Packet):
454     ack_id: int = 0
455     ack_bits: list[int | None] = [None for _ in range(ACK_BITS_SIZE)]
456
457     def __init__(
458         self,
459         version: int = VERSION,
460         packet_type: Type.ACK = Type.ACK,

```

```

461     flags: list[int] = [0 for _ in range(FLAGS_SIZE)],
462     sequence_id: int = None,
463     fragment_id: int | None = None,
464     fragment_number: int | None = None,
465     init_vector: int | None = None,
466     checksum: int | None = None,
467     ack_id: int = None,
468     ack_bits: list[int | None] = [None for _ in range(ACK_BITS_SIZE)],
469     data: bytes | None = None,
470 ) -> None:
471     super().__init__(
472         version,
473         Type.ACK,
474         flags,
475         sequence_id,
476         fragment_id,
477         fragment_number,
478         init_vector,
479         checksum,
480         data,
481     )
482     self.ack_id = ack_id
483     self.ack_bits = ack_bits
484
485     # dunder
486     def __str__(self) -> str:
487         s = self.pack(self)
488         data = self.data if self.data is not None else b""
489         pSize = len(s)
490         dSize = len(data)
491         if len(data) > 12:
492             data = f"{data[:11]}...{str(data[-1:])[1:]}"
493         return f"<{self.version}:{self.packet_type.name}"
494             {self.sequence_id}:{self.ack_id} {' '.join(map(str,self.flags))}
495             {data} [{pSize}:{dSize}]>"
496
497     # encode / decode
498     @staticmethod
499     def encodeAckId(ack_id: int) -> bytes:
500         return struct.pack("!I", ack_id)
501
502     @staticmethod
503     def decodeAckId(ack_id: bytes) -> int:
504         return struct.unpack("!I", ack_id)[0]

```

```

504 @staticmethod
505 def encodeAckBits(ack_bits: list[int]) -> bytes:
506     return struct.pack(
507         "!I",
508         int(
509             "".join(
510                 map(str, (int(bit) if bit is not None else 0 for bit in
511                           ack_bits))
512             ),
513         2,
514     )
515
516 @staticmethod
517 def decodeAckBits(ack_bits: bytes) -> list[int]:
518     ack_bits = struct.unpack("!I", ack_bits)[0]
519     ack_bits = [(ack_bits >> i) & 1 for i in range(ACK_BITS_SIZE)]
520     ack_bits.reverse()
521     return ack_bits
522
523 @staticmethod
524 def encodeHeader(
525     version: int,
526     packet_type: Type,
527     flags: list[int],
528     sequence_id: int,
529     fragment_id: int | None = None,
530     fragment_number: int | None = None,
531     init_vector: int | None = None,
532     checksum: int | None = None,
533     ack_id: int = 0,
534     ack_bits: list[int | None] = [None for _ in range(ACK_BITS_SIZE)],
535 ) -> bytes:
536     header = Packet.encodeHeader(
537         version,
538         packet_type,
539         flags,
540         sequence_id,
541         fragment_id,
542         fragment_number,
543         init_vector,
544         checksum,
545     )
546     ack_id = AckPacket.encodeAckId(ack_id)
547     ack_bits = AckPacket.encodeAckBits(ack_bits)

```



```

548         return header + ack_id + ack_bits
549
550     @staticmethod
551     def decodeHeader(
552         header: bytes,
553     ) -> tuple[
554         int,
555         Type,
556         list[int],
557         int,
558         int | None,
559         int | None,
560         int | None,
561         int | None,
562         int,
563         list[int | None],
564         int,
565     ]:
566         *h, offset = Packet.decodeHeader(header)
567         ack_id = AckPacket.decodeAckId(header[offset : offset + 4])
568         offset += 4
569         ack_bits = AckPacket.decodeAckBits(header[offset : offset + 4])
570         offset += 4
571         return *h, ack_id, ack_bits, offset
572
573     # pack / unpack
574     @classmethod
575     def _packHeader(cls, p) -> bytes:
576         header = cls.encodeHeader(
577             p.version,
578             p.packet_type,
579             p.flags,
580             p.sequence_id,
581             p.fragment_id,
582             p.fragment_number,
583             p.init_vector,
584             p.checksum,
585             p.ack_id,
586             p.ack_bits,
587         )
588         return header
589
590
591 class AuthPacket(Packet):
592     _public_key_size: int | None = None

```

```

593 public_key: EllipticCurvePublicKey | None = None
594 _certificate_size: int | None = None
595 certificate: Certificate | None = None
596
597 def __init__(
598     self,
599     version: int = VERSION,
600     packet_type: Type = Type.AUTH,
601     flags: list[int] = [0 for _ in range(FLAGS_SIZE)],
602     sequence_id: int = None,
603     fragment_id: int | None = None,
604     fragment_number: int | None = None,
605     init_vector: int | None = None,
606     checksum: int | None = None,
607     public_key_size: int | None = None,
608     public_key: EllipticCurvePublicKey = None,
609     certificate_size: int | None = None,
610     certificate: Certificate | None = None,
611 ) -> None:
612     super().__init__(
613         version,
614         Type.AUTH,
615         flags,
616         sequence_id,
617         fragment_id,
618         fragment_number,
619         init_vector,
620         checksum,
621         data=None,
622     )
623     self.public_key_size = public_key_size
624     self.public_key = public_key
625     self.certificate_size = certificate_size
626     self.certificate = certificate
627
628 # setter / getter
629 @property
630 def public_key_size(self) -> int | None:
631     if self._public_key_size is None:
632         self.public_key_size = (
633             AuthPacket.getPublicKeyBytesSize(self.public_key)
634             if self.public_key is not None
635             else None
636         )
637     return self._public_key_size

```

```

638
639 @public_key_size.setter
640 def public_key_size(self, v: int | None) -> None:
641     self._public_key_size = v
642
643 @staticmethod
644 def getPublicKeyBytesSize(publicKey: EllipticCurvePublicKey) -> int:
645     return len(auth.getDerFromPublicEc(publicKey))
646
647 @property
648 def certificate_size(self) -> int | None:
649     if self._certificate_size is None:
650         self.certificate_size = (
651             self.getCertificateByteSize(self.certificate)
652             if self.certificate is not None
653             else None
654         )
655     return self._certificate_size
656
657 @certificate_size.setter
658 def certificate_size(self, v: int | None) -> None:
659     self._certificate_size = v
660
661 @staticmethod
662 def getCertificateByteSize(certificate: Certificate) -> int:
663     return len(auth.getDerFromCertificate(certificate))
664
665 # encode / decode
666 @staticmethod
667 def encodePublicKeySize(public_key_size: int) -> bytes:
668     return struct.pack("!B", public_key_size)
669
670 @staticmethod
671 def decodePublicKeySize(public_key_size: bytes) -> int:
672     return struct.unpack("!B", public_key_size)[0]
673
674 @staticmethod
675 def encodePublicKey(public_key: EllipticCurvePublicKey) -> bytes:
676     return auth.getDerFromPublicEc(public_key)
677
678 @staticmethod
679 def decodePublicKey(public_key: bytes) -> EllipticCurvePublicKey:
680     return auth.getPublicEcFromDer(public_key)
681
682 @staticmethod

```

```

683 def encodeCertificateSize(certificate_size: int) -> bytes:
684     return struct.pack("!H", certificate_size)
685
686 @staticmethod
687 def decodeCertificateSize(certificate_size: bytes) -> int:
688     return struct.unpack("!H", certificate_size)[0]
689
690 @staticmethod
691 def encodeCertificate(certificate: Certificate) -> bytes:
692     return auth.getDerFromCertificate(certificate)
693
694 @staticmethod
695 def decodeCertificate(certificate: bytes) -> Certificate:
696     return auth.getCertificateFromDer(certificate)
697
698 @staticmethod
699 def encodeHeader(
700     version: int,
701     packet_type: Type,
702     flags: list[int],
703     sequence_id: int,
704     fragment_id: int | None = None,
705     fragment_number: int | None = None,
706     init_vector: int | None = None,
707     checksum: int | None = None,
708     public_key_size: int | None = None,
709     public_key: EllipticCurvePublicKey | None = None,
710     certificate_size: int | None = None,
711     certificate: Certificate | None = None,
712 ) -> bytes:
713     header = Packet.encodeHeader(
714         version,
715         packet_type,
716         flags,
717         sequence_id,
718         fragment_id,
719         fragment_number,
720         init_vector,
721         checksum,
722     )
723     public_key_size = AuthPacket.encodePublicKeySize(public_key_size)
724     public_key = AuthPacket.encodePublicKey(public_key)
725     certificate_size = (
726         AuthPacket.encodeCertificateSize(certificate_size)
727         if certificate_size is not None

```

```

728         else b"""
729     )
730     certificate = (
731         AuthPacket.encodeCertificate(certificate)
732         if certificate is not None
733         else b"""
734     )
735     return header + public_key_size + public_key + certificate_size +
        certificate

736
737 @staticmethod
738 def decodeHeader(
739     header: bytes,
740 ) -> tuple[
741     int,
742     Type,
743     list[int],
744     int,
745     int | None,
746     int | None,
747     int | None,
748     int | None,
749     int,
750     EllipticCurvePublicKey,
751     int | None,
752     Certificate | None,
753     int,
754 ]:
755     *h, offset = Packet.decodeHeader(header)
756     public_key_size = AuthPacket.decodePublicKeySize(header[offset :
        offset + 1])
757     offset += 1
758     public_key = AuthPacket.decodePublicKey(
759         header[offset : offset + public_key_size]
760     )
761     offset += public_key_size
762     if offset < len(header): # check if more bytes left to decode
763         certificate_size = AuthPacket.decodeCertificateSize(
764             header[offset : offset + 2]
765         )
766         offset += 2
767         certificate = AuthPacket.decodeCertificate(
768             header[offset : offset + certificate_size]
769         )
770         offset += certificate_size

```

```

771         else:
772             certificate_size = None
773             certificate = None
774         return *h, public_key_size, public_key, certificate_size,
            certificate, offset
775
776     # pack / unpack
777     @classmethod
778     def _packHeader(cls, p) -> bytes:
779         header = cls.encodeHeader(
780             p.version,
781             p.packet_type,
782             p.flags,
783             p.sequence_id,
784             p.fragment_id,
785             p.fragment_number,
786             p.init_vector,
787             p.checksum,
788             p.public_key_size,
789             p.public_key,
790             p.certificate_size,
791             p.certificate,
792         )
793         return header
794
795     @classmethod
796     def unpack(cls, bytesP: bytes):
797         *header, offset = cls._unpackHeader(bytesP)
798         return cls(*header)
799
800
801 class HeartbeatPacket(Packet):
802     heartbeat: bool
803
804     def __init__(
805         self,
806         version: int = VERSION,
807         packet_type: Type = Type.HEARTBEAT,
808         flags: list[int] = [0 for _ in range(FLAGS_SIZE)],
809         sequence_id: int = None,
810         fragment_id: int | None = None,
811         fragment_number: int | None = None,
812         init_vector: int | None = None,
813         checksum: int | None = None,
814         heartbeat: bool = 0,

```

```

815         data: bytes | None = None,
816     ) -> None:
817         super().__init__(
818             version,
819             Type.HEARTBEAT,
820             flags,
821             sequence_id,
822             fragment_id,
823             fragment_number,
824             init_vector,
825             checksum,
826             data,
827         )
828         self.heartbeat = heartbeat
829
830     # encode / decode
831     @staticmethod
832     def encodeHeartbeat(heartbeat: bool) -> bytes:
833         return struct.pack("!?", heartbeat)
834
835     @staticmethod
836     def decodeHeartbeat(heartbeat: bytes) -> bool:
837         return struct.unpack("!?", heartbeat)[0]
838
839     @staticmethod
840     def encodeHeader(
841         version: int,
842         packet_type: Type,
843         flags: list[int],
844         sequence_id: int,
845         fragment_id: int | None = None,
846         fragment_number: int | None = None,
847         init_vector: int | None = None,
848         checksum: int | None = None,
849         heartbeat: bool = 0,
850     ) -> bytes:
851         header = Packet.encodeHeader(
852             version,
853             packet_type,
854             flags,
855             sequence_id,
856             fragment_id,
857             fragment_number,
858             init_vector,
859             checksum,

```

```

860     )
861     heartbeat = HeartbeatPacket.encodeHeartbeat(heartbeat)
862     return header + heartbeat
863
864     @staticmethod
865     def decodeHeader(
866         header: bytes,
867     ) -> tuple[
868         int,
869         Type,
870         list[int],
871         int,
872         int | None,
873         int | None,
874         int | None,
875         int | None,
876         bool,
877         int,
878     ]:
879         *h, offset = Packet.decodeHeader(header)
880         heartbeat = HeartbeatPacket.decodeHeartbeat(header[offset : offset +
881             1])
881         offset += 1
882         return *h, heartbeat, offset
883
884     # pack / unpack
885     @classmethod
886     def _packHeader(cls, p) -> bytes:
887         header = cls.encodeHeader(
888             p.version,
889             p.packet_type,
890             p.flags,
891             p.sequence_id,
892             p.fragment_id,
893             p.fragment_number,
894             p.init_vector,
895             p.checksum,
896             p.heartbeat,
897         )
898         return header
899
900
901 class ErrorPacket(Packet):
902     _major: error.Major
903     _minor: error.Minor

```



```

904
905 def __init__(
906     self,
907     version: int = VERSION,
908     packet_type: Type = Type.ERROR,
909     flags: list[int] = [0 for _ in range(FLAGS_SIZE)],
910     sequence_id: int = None,
911     fragment_id: int | None = None,
912     fragment_number: int | None = None,
913     init_vector: int | None = None,
914     checksum: int | None = None,
915     major: error.Major | int = error.Major.ERROR,
916     minor: error.Minor | int = 0,
917     data: bytes | None = None,
918 ) -> None:
919     super().__init__(
920         version,
921         Type.ERROR,
922         flags,
923         sequence_id,
924         fragment_id,
925         fragment_number,
926         init_vector,
927         checksum,
928         data,
929     )
930     self.major = major
931     self.minor = minor
932
933 @property
934 def major(self) -> error.Major:
935     return self._major
936
937 @major.setter
938 def major(self, v: error.Major | int):
939     if isinstance(v, error.Major):
940         self._major = v
941     else:
942         self._major = error.Major(v)
943
944 @property
945 def minor(self) -> error.Minor:
946     return self._minor
947
948 @minor.setter

```

```

949 def minor(self, v: error.Minor | int):
950     if isinstance(v, error.Minor):
951         self._minor = v
952     else:
953         self._minor = error.getMinor(self.major, v)
954
955     # dunder
956 def __str__(self) -> str:
957     s = self.pack(self)
958     data = self.data if self.data is not None else b""
959     pSize = len(s)
960     dSize = len(data)
961     return f"<{self.version}:{self.packet_type.name} {self.sequence_id}
        {''.join(map(str,self.flags))}
        {self.major.name}.{self.minor.name}: {data} [{pSize}:{dSize}]>"
962
963     # encode / decode
964 @staticmethod
965 def _encodeMajor(major: error.Major) -> int:
966     return major.value
967
968 @staticmethod
969 def _decodeMajor(major: int) -> error.Major:
970     return error.Major(major)
971
972 @staticmethod
973 def _encodeMinor(minor: error.Minor) -> int:
974     return minor.value if minor != error.Minor else 0
975
976 @staticmethod
977 def _decodeMinor(major: error.Major, minor: int) -> error.Minor:
978     return error.getMinor(major, minor)
979
980 def encodeMajorMinor(major: int, minor: int) -> bytes:
981     majorMinor = (ErrorPacket._encodeMajor(major) * 16) |
        ErrorPacket._encodeMinor(
982         minor
983     )
984     return struct.pack("!B", majorMinor)
985
986 def decodeMajorMinor(majorMinor: bytes) -> tuple[int, int]:
987     majorMinor = struct.unpack("!B", majorMinor)[0]
988     major = ErrorPacket._decodeMajor(majorMinor >> 4)
989     minor = ErrorPacket._decodeMinor(major, majorMinor & 15)
990     return major, minor

```

```

991
992 @staticmethod
993 def encodeHeader(
994     version: int,
995     packet_type: Type,
996     flags: list[int],
997     sequence_id: int,
998     fragment_id: int | None = None,
999     fragment_number: int | None = None,
1000     init_vector: int | None = None,
1001     checksum: int | None = None,
1002     major: int = 0,
1003     minor: int = 0,
1004 ) -> bytes:
1005     header = Packet.encodeHeader(
1006         version,
1007         packet_type,
1008         flags,
1009         sequence_id,
1010         fragment_id,
1011         fragment_number,
1012         init_vector,
1013         checksum,
1014     )
1015     majorMinor = ErrorPacket.encodeMajorMinor(major, minor)
1016     return header + majorMinor
1017
1018 @staticmethod
1019 def decodeHeader(
1020     header: bytes,
1021 ) -> tuple[
1022     int,
1023     Type,
1024     list[int],
1025     int,
1026     int | None,
1027     int | None,
1028     int | None,
1029     int | None,
1030     int,
1031     int,
1032     int,
1033 ]:
1034     *h, offset = Packet.decodeHeader(header)
1035     major, minor = ErrorPacket.decodeMajorMinor(header[offset : offset +

```

```

        1])
1036         offset += 1
1037         return *h, major, minor, offset
1038
1039     # pack / unpack
1040     @classmethod
1041     def _packHeader(cls, p) -> bytes:
1042         header = cls.encodeHeader(
1043             p.version,
1044             p.packet_type,
1045             p.flags,
1046             p.sequence_id,
1047             p.fragment_id,
1048             p.fragment_number,
1049             p.init_vector,
1050             p.checksum,
1051             p.major,
1052             p.minor,
1053         )
1054         return header
1055
1056
1057 def unpack(rawP:bytes) -> Packet:
1058     packet_type = Packet.decodeVersionType(rawP[0:1])[1]
1059     match packet_type:
1060         case Type.DEFAULT:
1061             return Packet.unpack(rawP)
1062         case Type.ACK:
1063             return AckPacket.unpack(rawP)
1064         case Type.AUTH:
1065             return AuthPacket.unpack(rawP)
1066         case Type.HEARTBEAT:
1067             return HeartbeatPacket.unpack(rawP)
1068         case Type.ERROR:
1069             return ErrorPacket.unpack(rawP)
1070         case _:
1071             logger.warning(f"Cannot unpack '{packet_type}' due to invalid
                             packet type.")

```

```

1 # udp.server
2 import base64
3 import json
4 import socket

```

```

5 import time
6 from datetime import datetime
7 from threading import Event, Lock, Thread
8
9 import requests
10 from cryptography.hazmat.primitives.asymmetric.rsa import RSAPrivateKey
11
12 from . import (
13     HEARTBEAT_MAX_TIME,
14     HEARTBEAT_MIN_TIME,
15     MAX_CLIENTS,
16     auth,
17     bcolors,
18     error,
19     logger,
20     node,
21     packet,
22 )
23
24
25 class Server(node.Node):
26     clients: dict[tuple[str, int], node.Node]
27     clientsLock: Lock
28     clientDeleteEvent: Event
29     rsaKey: RSAPrivateKey | None
30     heartbeatThread: Thread
31     onClientJoin: None
32     onClientLeave: None
33     maxClients: int
34
35     def __init__(
36         self,
37         addr,
38         maxClients: int = MAX_CLIENTS,
39         rsaKey: RSAPrivateKey | None = None,
40         onClientJoin=None,
41         onClientLeave=None,
42         onReceiveData=None,
43     ) -> None:
44         self.clients = {}
45         self.clientsLock = Lock()
46         self.clientDeleteEvent = Event()
47         self.clientDeleteEvent.set()
48         self.rsaKey = rsaKey if rsaKey is not None else auth.generateRsaKey()
49         self.onClientJoin = onClientJoin

```

```

50     self.onClientLeave = onClientLeave
51     self.maxClients = maxClients
52     s = socket.socket(type=socket.SOCK_DGRAM)
53     super().__init__(
54         addr,
55         cert=auth.generateUserCertificate(self.rsaKey),
56         socket=s,
57         onReceiveData=onReceiveData,
58     )
59     self.heartbeatThread = Thread(
60         name=f"{self.port}:Heartbeat", target=self.heartbeat, daemon=True
61     )
62     self.bind(self.addr)
63
64     def receiveAck(self, p: packet.AckPacket, addr: tuple[str, int]) -> None:
65         super().receiveAck(p, addr)
66         if p.data is not None and not self.getHandshake(
67             addr
68         ): # ack has payload & client has not completed handshake =>
           validate handshake
69             if not self.validateHandshake(addr, p.data):
70                 # raise ValueError(f"Local finished value does not match peer
           finished value {p.data}")
71                 logger.error(
72                     f"Local finished value does not match peer finished value
73                         {p.data}"
74                 )
75                 self.queueError(
76                     addr,
77                     major=error.Major.CONNECTION,
78                     minor=error.ConnectionErrorCodes.FINISH_INVALID,
79                     data=b"Invalid finish.",
80                 )
81             else:
82                 # print(f"{bcolors.OKGREEN}# Handshake with {addr}
           successful.{bcolors.ENDC}")
83                 logger.info(
84                     f"{bcolors.OKGREEN}# Handshake with {addr}
85                         successful.{bcolors.ENDC}"
86                 )
87                 if self.onClientJoin:
88                     self.onClientJoin(addr, self.getClientId(addr))
89
90     def receiveAuth(
91         self, p: packet.AuthPacket, addr: tuple[str, int]

```

```

90 ) -> tuple[packet.AuthPacket, tuple[str, int]]:
91     if addr not in self.clients: # new client
92         if self.isNotFull(): # check space
93             # print(f"{bcolors.WARNING}# Handshake with {addr}
94                 starting.{bcolors.ENDC}")
95             logger.info(
96                 f"{bcolors.WARNING}# Handshake with {addr}
97                     starting.{bcolors.ENDC}"
98             )
99             valid, accountId = self.validateCertificate(p.certificate)
100             if not valid:
101                 # raise ValueError(f"Invalid peer cert {p.certificate}")
102                 logger.error(f"Invalid peer cert {p.certificate}")
103                 self.queueError(
104                     addr,
105                     major=error.Major.CONNECTION,
106                     minor=error.ConnectionErrorCodes.CERTIFICATE_INVALID,
107                     data=b"Invalid Certificate.",
108                 )
109             else:
110                 self.makeClient(addr, p.certificate, accountId)
111                 self.regenerateEcKey(addr)
112                 sessionKey = auth.generateSessionKey(
113                     self.getEcKey(addr), p.public_key
114                 )
115                 self.setSessionKey(addr, sessionKey)
116                 self.queueAuth(addr, self.cert,
117                     self.getEcKey(addr).public_key())
118                 self.queueFinished(addr, p.sequence_id,
119                     self.getSessionKey(addr))
120             else:
121                 # print(f"{bcolors.FAIL}# Handshake with {addr} denied due to
122                     NO_SPACE.{bcolors.ENDC}")
123                 logger.warning(
124                     f"{bcolors.FAIL}# Handshake with {addr} denied due to
125                         NO_SPACE.{bcolors.ENDC}"
126                 )
127                 self.queueError(
128                     addr,
129                     major=error.Major.CONNECTION,
130                     minor=error.ConnectionErrorCodes.NO_SPACE,
131                     data=b"Server is Full.",
132                 )
133             else:
134                 sessionKey = auth.generateSessionKey(self.getEcKey(addr),

```

```

        p.public_key)
129     if addr in self.clients:
130         if self.getSessionKey(addr) != sessionKey: # new client
            sessionKey
131         # print(f"{bcolors.WARNING}# Handshake with {addr}
            reset.{bcolors.ENDC}")
132         logger.info(
133             f"{bcolors.WARNING}# Handshake with {addr}
                reset.{bcolors.ENDC}"
134         )
135         valid, accountId = self.validateCertificate(p.certificate)
136         if not valid:
137             # raise ValueError(f"Invalid peer cert {p.certificate}")
138             logger.warning(f"Invalid peer cert {p.certificate}")
139             self.queueError(
140                 addr,
141                 major=error.Major.CONNECTION,
142                 minor=error.ConnectionErrorCodes.CERTIFICATE_INVALID,
143                 data=b"Invalid Certificate.",
144             )
145         else:
146             self.regenerateEcKey(addr)
147             # self.clients[addr].cert = p.certificate # shouldn't
            change
148             sessionKey = auth.generateSessionKey(
149                 self.getEcKey(addr), p.public_key
150             )
151             self.setSessionKey(addr, sessionKey) # make new session
            key
152             self.queueAuth(addr, self.cert,
                self.getEcKey(addr).public_key())
153             self.queueFinished(addr, p.sequence_id,
                self.getSessionKey(addr))
154         return (p, addr)
155
156     def queueDisconnect(
157         self,
158         flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
159         data: bytes | None = None,
160     ):
161         with self.clientsLock:
162             clientAddrs = [addr for addr in self.clients]
163         for addr in clientAddrs:
164             self.queueError(
165                 addr,

```



```

166         flags=flags,
167         major=error.Major.DISCONNECT,
168         minor=error.DisconnectErrorCodes.SERVER_DISCONNECT,
169         data=data,
170     )
171
172     def getSessionKey(self, clientAddr: tuple[str, int]) -> bytes | None:
173         with self.clientsLock:
174             return self.clients[clientAddr].sessionKey
175
176     def setSessionKey(self, clientAddr: tuple[str, int], sessionKey: bytes)
177         -> None:
178         with self.clientsLock:
179             self.clients[clientAddr].sessionKey = sessionKey
180
181     def getHandshake(self, clientAddr: tuple[str, int]) -> bool:
182         with self.clientsLock:
183             return self.clients[clientAddr].handshake
184
185     def getSentAckBit(self, clientAddr: tuple[str, int], p: packet.Packet) ->
186         bool:
187         with self.clientsLock:
188             return self.clients[clientAddr].sentAckBits[p.sequence_id]
189
190     def setSentAckBit(self, clientAddr: tuple[str, int], ackBit: int, v:
191         bool) -> None:
192         with self.clientsLock:
193             self.clients[clientAddr].sentAckBits[ackBit] = v
194
195     def getSentAckBits(self, clientAddr: tuple[str, int]) -> list[bool]:
196         with self.clientsLock:
197             return self.clients[clientAddr].sentAckBits
198
199     def getRecvAckBit(self, clientAddr: tuple[str, int], p: packet.Packet) ->
200         bool:
201         with self.clientsLock:
202             return self.clients[clientAddr].recvAckBits[p.sequence_id]
203
204     def getRecvAckBits(self, clientAddr: tuple[str, int]) -> list[bool]:
205         with self.clientsLock:
206             return self.clients[clientAddr].recvAckBits
207
208     def setRecvAckBit(self, clientAddr: tuple[str, int], ackBit: int, v:
209         bool) -> None:
210         with self.clientsLock:

```

```

206         self.clients[clientAddr].recvAckBits[ackBit] = v
207
208     def getNewestSeqId(self, clientAddr: tuple[str, int]) -> int:
209         with self.clientsLock:
210             if clientAddr in self.clients:
211                 return self.clients[clientAddr].newestSeqId
212             else:
213                 return 0
214
215     def setNewestSeqId(self, clientAddr: tuple[str, int], newSeqId: int) ->
None:
216         with self.clientsLock:
217             if clientAddr in self.clients:
218                 self.clients[clientAddr].newestSeqId = newSeqId
219
220     def getFragBuffer(
221         self, clientAddr: tuple[str, int]
222     ) -> dict[int, list[packet.Packet]]:
223         with self.clientsLock:
224             return self.clients[clientAddr].fragBuffer
225
226     def getEcKey(self, clientAddr: tuple[str, int]) ->
auth.ec.EllipticCurvePrivateKey:
227         with self.clientsLock:
228             return self.clients[clientAddr].ecKey
229
230     def getSequenceId(self, clientAddr: tuple[str, int]) -> int | None:
231         with self.clientsLock:
232             return (
233                 self.clients[clientAddr].sequenceId
234                 if clientAddr in self.clients
235                 else None
236             )
237
238     def getQueue(self, clientAddr: tuple[str, int]) -> node.Queue:
239         with self.clientsLock:
240             return (
241                 self.clients[clientAddr].queue
242                 if clientAddr in self.clients
243                 else self.queue
244             )
245
246     def getSequenceIdLock(self, clientAddr: tuple[str, int]) -> Lock:
247         with self.clientsLock:
248             return self.clients[clientAddr].sequenceIdLock

```

```

249
250 def incrementSequenceId(self, clientAddr: tuple[str, int]) -> None:
251     with self.getSequenceIdLock(clientAddr):
252         with self.clientsLock:
253             self.clients[clientAddr].sequenceId += 1
254
255 def getHeartbeat(self, clientAddr: tuple[str, int]) -> datetime:
256     with self.clientsLock:
257         return self.clients[clientAddr].heartbeat
258
259 def setHeartbeat(self, clientAddr: tuple[str, int], v: datetime) -> None:
260     with self.clientsLock:
261         self.clients[clientAddr].heartbeat = v
262
263 def regenerateEcKey(self, clientAddr: tuple[str, int]) -> None:
264     with self.clientsLock:
265         self.clients[clientAddr].regenerateEcKey()
266
267 def checkClientExists(self, clientAddr: tuple[str, int]) -> bool:
268     with self.clientsLock:
269         return clientAddr in self.clients
270
271 def validateHandshake(self, clientAddr: tuple[str, int], finished: bytes)
272     -> bool:
273     with self.clientsLock:
274         return self.clients[clientAddr].validateHandshake(finished)
275
276 def getClientLength(self) -> int:
277     with self.clientsLock:
278         return len(self.clients)
279
280 def getClientId(self, clientAddr: tuple[str, int]) -> int:
281     with self.clientsLock:
282         return self.clients[clientAddr].accountId
283
284 def getClientIds(self) -> list[int]:
285     with self.clientsLock:
286         return [client.id for addr, client in self.clients.items()]
287
288 def isNotFull(self) -> bool:
289     with self.clientsLock:
290         return len(self.clients) < self.maxClients # check space
291
292 def isEmpty(self) -> bool:
293     with self.clientsLock:

```

```

293         return len(self.clients) == 0
294
295     def listen(self) -> None:
296         logger.info(
297             f"{bcolors.HEADER}Listening @
                {self.socket.getsockname()}{bcolors.ENDC}"
298         )
299         while self.isRunning.is_set():
300             p, addr = self.receivePacket()
301             if p is not None and addr is not None:
302                 if self.checkClientExists(addr): # client exists
303                     self.setHeartbeat(addr, datetime.now())
304                     if self.getHandshake(
305                         addr
306                     ): # client handshake complete => allow all packet types
307                         self.receive(p, addr)
308                     else:
309                         if (
310                             p.packet_type
311                             in (packet.Type.AUTH, packet.Type.ACK,
312                                 packet.Type.ERROR)
313                         ): # client handshake incomplete => drop all
                            non-AUTH | non-ACK | non-ERROR packets
314                             self.receive(p, addr)
315                         else:
316                             if p.packet_type in (
317                                 packet.Type.AUTH,
318                                 packet.Type.ERROR,
319                             ): # client not exists => drop all non-AUTH | non-ERROR
                                    packets
320                                 self.receive(p, addr)
321                             else:
322                                 logger.warning(
323                                     f"{bcolors.WARNING}! {addr} :{bcolors.ENDC}
                                        {bcolors.WARNING}{p}{bcolors.ENDC}"
324                                 )
325                         else:
326                             logger.info("| listen thread stopping...")
327
328     def heartbeat(self) -> None:
329         while self.isRunning.is_set():
330             time.sleep(HEARTBEAT_MIN_TIME)
331             with self.clientsLock:
332                 clients = [k for k in self.clients.keys()]
333                 for clientAddr in clients:

```

```

333         heartbeat = self.getHeartbeat(clientAddr)
334         delta = (datetime.now() - heartbeat).seconds
335         if delta > HEARTBEAT_MAX_TIME:
336             self.removeClient(
337                 clientAddr,
338                 debugStr=f"due to heartbeat timeout (last contact was
339                     {heartbeat})",
340             )
341         elif delta > HEARTBEAT_MIN_TIME:
342             self.queueHeartbeat(clientAddr, heartbeat=False)
343     else:
344         logger.info("| heartbeat thread stopping...")
345
346     def makeClient(
347         self, clientAddr: tuple[str, int], cert: auth.x509.Certificate,
348         accountId: int
349     ) -> None:
350         c = node.Node(
351             clientAddr,
352             cert=cert,
353             accountId=accountId,
354             sendLock=self.sendLock,
355             socket=self.socket,
356         )
357         c.outboundThread.start()
358         with self.clientsLock:
359             self.clients[clientAddr] = c
360
361     def removeClient(self, clientAddr: tuple[str, int], debugStr="") -> None:
362         if self.checkClientExists(clientAddr):
363             cId = self.getClientId(clientAddr)
364             with self.clientsLock:
365                 logger.info(
366                     f"{bcolors.FAIL}# Client {clientAddr} was removed{'
367                         '+debugStr}.{bcolors.ENDC}"
368                 )
369                 self.clients[clientAddr].isRunning.clear()
370                 del self.clients[clientAddr]
371                 if self.onClientLeave:
372                     self.onClientLeave(clientAddr, cId)
373
374     # misc
375     def startThreads(self) -> None:
376         super().startThreads()
377         self.heartbeatThread.start()

```

```

375
376 def validateCertificate(self, certificate: auth.x509.Certificate) -> bool:
377     url = f"http://{self.host}:5000/auth/certificate/validate"
378     headers = {"Content-Type": "application/json"}
379     certificate = base64.encodebytes(
380         auth.getDerFromCertificate(certificate)
381     ).decode()
382     data = {"certificate": certificate}
383     try:
384         r = requests.get(url, headers=headers, data=json.dumps(data))
385         if r.status_code == 200:
386             return r.json()["valid"], r.json()["account-id"]
387         else:
388             return False
389     except: # noqa: E722
390         # Cert server unresponsive
391         return False
392
393 def quit(
394     self, msg: str = "quit call", e: Exception | None = None
395 ) -> Exception | None:
396     self.queueDisconnect(data=msg.encode())
397     self.queue.join()
398     e = super().quit(msg, e)
399     if self.heartbeatThread.is_alive:
400         self.heartbeatThread.join()
401     return e
402
403 def handleDisconnectError(
404     self, p: packet.ErrorPacket, addr: tuple[str, int], e:
405         error.DisconnectError
406 ) -> None:
407     match e:
408         case error.ServerDisconnectError():
409             pass # should not react to server disconnect
410         case error.ClientDisconnectError():
411             self.removeClient(addr, "The client has closed")
412         case _:
413             raise e

```

```

1 # udp.utils
2 import zlib
3

```

```

4 from . import MAX_FRAGMENT_SIZE
5
6
7 def compressData(data: bytes) -> bytes:
8     # default speed
9     # no header or checksum
10    return zlib.compress(data, -1, -15)
11
12
13 def decompressData(data: bytes) -> bytes:
14     # no header or checksum
15    return zlib.decompress(data, -15)
16
17
18 def generateChecksum(data: bytes) -> int:
19    return zlib.crc32(data)
20
21
22 def fragmentData(data: bytes) -> list[bytes]:
23    return [
24        data[i : i + MAX_FRAGMENT_SIZE] for i in range(0, len(data),
25            MAX_FRAGMENT_SIZE)
26    ]
27
28 def defragmentData(fragments: list[bytes]) -> bytes:
29    return b"".join(fragments)

```

9.3.2 server

```

1 # server.__init__
2 import os
3
4 import dotenv
5 from flask import Flask
6
7 from udp import logger # noqa: F401
8
9 from .models import * # noqa: F403
10
11 from sqlalchemy_utils import database_exists, create_database
12
13 dotenv.load_dotenv()

```

```

14 PRUNE_TIME = int(os.environ.get("PRUNE_TIME"))
15
16
17 def create_app():
18     app = Flask(__name__)
19
20     app.jinja_env.trim_blocks = True
21     app.jinja_env.lstrip_blocks = True
22
23     app.config["SECRET_KEY"] = os.environ.get("SECRET_KEY").encode()
24     uri = os.environ.get("SQLALCHEMY_DATABASE_URI")
25     _init = False
26     if not database_exists(uri):
27         _init = True
28         create_database(uri)
29     app.config["SQLALCHEMY_DATABASE_URI"] = uri
30
31     db.init_app(app)  # noqa: F405
32
33     with app.app_context():
34         db.create_all()  # noqa: F405
35
36     if _init:
37         with app.app_context():
38             # init games
39             from rps import ID, NAME, MIN_PLAYERS, MAX_PLAYERS
40             Statement.createGame(ID, NAME, MIN_PLAYERS, MAX_PLAYERS)  # noqa:
41                                     F405
42             # example accounts
43             m = Statement.createAccount("Mario", "ItsAMe123")  # noqa: F405
44             p = Statement.createAccount("Peach", "MammaMia!")  # noqa: F405
45             b = Statement.createAccount("Bowser", "M4r10SucK5")  # noqa: F405
46             Statement.createFriends(m.id, p.id)  # noqa: F405
47             Statement.createFriends(p.id, b.id)  # noqa: F405
48
49     from .main import main as main_blueprint
50
51     app.register_blueprint(main_blueprint)
52
53     return app

```

```

1 # server.lobbies
2 import os

```



```

3
4 import dotenv
5 from flask import Flask
6
7 from udp import logger # noqa: F401
8
9 from .models import * # noqa: F403
10
11 from sqlalchemy_utils import database_exists, create_database
12
13 dotenv.load_dotenv()
14 PRUNE_TIME = int(os.environ.get("PRUNE_TIME"))
15
16
17 def create_app():
18     app = Flask(__name__)
19
20     app.jinja_env.trim_blocks = True
21     app.jinja_env.lstrip_blocks = True
22
23     app.config["SECRET_KEY"] = os.environ.get("SECRET_KEY").encode()
24     uri = os.environ.get("SQLALCHEMY_DATABASE_URI")
25     _init = False
26     if not database_exists(uri):
27         _init = True
28         create_database(uri)
29     app.config["SQLALCHEMY_DATABASE_URI"] = uri
30
31     db.init_app(app) # noqa: F405
32
33     with app.app_context():
34         db.create_all() # noqa: F405
35
36     if _init:
37         with app.app_context():
38             # init games
39             from rps import ID, NAME, MIN_PLAYERS, MAX_PLAYERS
40             Statement.createGame(ID, NAME, MIN_PLAYERS, MAX_PLAYERS) # noqa:
41                               F405
42             # example accounts
43             m = Statement.createAccount("Mario", "ItsAMe123") # noqa: F405
44             p = Statement.createAccount("Peach", "MammaMia!") # noqa: F405
45             b = Statement.createAccount("Bowser", "M4r10SucK5") # noqa: F405
46             Statement.createFriends(m.id, p.id) # noqa: F405
47             Statement.createFriends(p.id, b.id) # noqa: F405

```

```

47
48     from .main import main as main_blueprint
49
50     app.register_blueprint(main_blueprint)
51
52     return app

```

```

1  # server.main
2  import atexit
3  import base64
4
5  from flask import (
6      Blueprint,
7      abort,
8      g,
9      jsonify,
10     request,
11 )
12 from flask_httpauth import HTTPBasicAuth
13
14 import udp.auth
15
16 from . import Statement
17 from .lobbies import LobbyHandler
18
19 main = Blueprint("main", __name__)
20 auth = HTTPBasicAuth()
21 rsaKey = udp.auth.generateRsaKey()
22 lobbyHandler = LobbyHandler(rsaKey=rsaKey)
23
24
25 def quit() -> None:
26     lobbyHandler.quit()
27
28
29 atexit.register(quit)
30
31
32 @auth.verify_password
33 def verifyPassword(username: str, password: str) -> bool:
34     account = Statement.validateToken(username) # check token
35     if not account: # if token not valid
36         account = Statement.findAccount(username=username) # check account

```

```

37         if not account or not account.verifyPassword(
38             password
39         ): # if account not exist or wrong password
40             return False
41     g.account = account
42     return True
43
44
45 # Index
46 @main.route("/")
47 def index():
48     return jsonify({})
49
50
51 # auth
52 @main.route("/auth/register", methods=["POST"])
53 def createAccount():
54     username = request.json.get("username")
55     password = request.json.get("password")
56     if not (username or password): # check not null
57         abort(400) # missing args
58     if Statement.findAccount(username): # check if account exists
59         abort(400) # account already exists
60     account = Statement.createAccount(username, password)
61     return jsonify({"account-id": account.id, "username": account.username}),
    201
62
63
64 @main.route("/auth/token")
65 @auth.login_required
66 def getAuthToken():
67     return jsonify({"token": g.account.generateToken()})
68
69
70 @main.route("/auth/key")
71 @auth.login_required
72 def getKey():
73     return jsonify(
74         {
75             "key": base64.encodebytes(g.account.private_key).decode(),
76             "account-id": g.account.id,
77         }
78     )
79
80

```

```

81 @main.route("/auth/certificate")
82 @auth.login_required
83 def getCert():
84     # return server certificate
85     return None
86
87
88 @main.route("/auth/certificate/validate")
89 def validateCert():
90     valid = False
91     certificate = request.json.get("certificate")
92     certificate = base64.decodebytes(certificate.encode())
93     if certificate is not None:
94         certificate = udp.auth.getCertificateFromDer(certificate)
95         attributes = udp.auth.getUserCertificateAttributes(certificate)
96         if attributes["account-id"] is not None:
97             account = Statement.getAccount(attributes["account-id"])
98             publicKey = udp.auth.getRsaPublicFromDer(account.public_key)
99         else:
100             publicKey = rsaKey.public_key()
101             valid = udp.auth.validateCertificate(certificate, publicKey)
102         return jsonify({"valid": valid, "account-id":
103             attributes["account-id"]})
104     else:
105         abort(400) # missing args
106
107 @main.route("/auth/test")
108 @auth.login_required
109 def authTest():
110     return jsonify({"hello": g.account.username})
111
112
113 # game
114 @main.route("/games/")
115 @auth.login_required
116 def getGames():
117     return jsonify({game.id: game.name for game in Statement.getGames()})
118
119
120 @main.route("/lobby/all")
121 @auth.login_required
122 def getLobbies():
123     lobbies = LobbyHandler.getAll()
124     games = {game.id: game.name for game in Statement.getGames()}

```

```

125     data = lambda lobby: { # noqa: E731
126         "game": {"game-id": lobby.game_id, "game-name": games[lobby.game_id]},
127         "size": Statement.getLobbySize(lobby.id),
128         "is-full": Statement.getIsLobbyFree(lobby.id),
129     }
130     return jsonify({lobby.id: data(lobby) for lobby in lobbies})
131
132
133 @main.route("/lobby/create", methods=["POST"])
134 @auth.login_required
135 def createLobby():
136     gameId = request.json.get("game-id")
137     gameName = request.json.get("game-name")
138     if not (gameId or gameName): # check args
139         abort(400) # missing args
140     game = None
141     if gameId: # check gameId not null
142         game = Statement.getGame(gameId)
143     if not game: # check gameId null
144         if gameName: # check gameName not null
145             game = Statement.findGame(gameName)
146     if not game: # check game null
147         abort(404) # no game found
148     addr = _getAddr()
149     lobby = lobbyHandler.createLobby(addr, game.id)
150     return jsonify(
151         {"lobby-id": lobby.id, "lobby-addr": lobby.getAddr(), "game-id":
152             lobby.gameId}
153     ), 201
154
155 def _getAddr():
156     host, port = request.host.split(":")
157     port = int(port)
158     return (host, port)
159
160
161 @main.route("/lobby/")
162 @auth.login_required
163 def getLobby():
164     lobbyId = request.json.get("lobby-id")
165     if not lobbyId:
166         abort(400) # missing args
167     lobby = lobbyHandler.getLobby(lobbyId)
168     return jsonify(

```

```

169         {"lobby-id": lobby.id, "lobby-addr": lobby.getAddr(), "game-id":
            lobby.gameId}
170     )
171
172
173 @main.route("/lobby/members")
174 @auth.login_required
175 def getMembers():
176     return jsonify(lobbyHandler.getMembers)
177
178
179 @main.route("/lobby/find")
180 @auth.login_required
181 def findLobby():
182     gameId = request.json.get("game-id")
183     gameName = request.json.get("game-name")
184     if not (gameId or gameName): # check args
185         abort(400) # missing args
186     game = None
187     if gameId: # check gameId not null
188         game = Statement.getGame(gameId)
189     if not game: # check gameId null
190         if gameName: # check gameName not null
191             game = Statement.findGame(gameName)
192     if not game: # check game null
193         abort(404) # no game found
194     lobby = lobbyHandler.findLobbies(game.id)
195     lobby = lobby[0] if len(lobby) > 0 else None
196     if lobby is not None:
197         return jsonify(
198             {
199                 "lobby-id": lobby.id,
200                 "lobby-addr": lobby.getAddr(),
201                 "game-id": lobby.gameId,
202             }
203         )
204     else:
205         abort(404)
206
207
208 @main.route("/friends/")
209 @auth.login_required
210 def getFriends():
211     friends = Statement.getFriends(g.account.id)
212     return jsonify(

```

```

213     {
214         "friends": [
215             {"id": account.id, "username": account.username} for account
                in friends
216         ]
217     }
218 )
219
220
221 @main.route("/friends/add", methods=["POST"])
222 @auth.login_required
223 def addFriend():
224     username = request.json.get("username")
225     if username is None:
226         abort(400) # missing args
227     account = g.account
228     other = Statement.findAccount(username)
229     if other is None:
230         abort(404)
231     Statement.createFriends(account.id, other.id)
232     return jsonify(
233         {
234             "account": {"id": account.id, "username": account.username},
235             "other": {"id": other.id, "username": other.username},
236         }
237     ), 201
238
239
240 @main.route("/friend/remove", methods=["DELETE"])
241 @auth.login_required
242 def removeFriend():
243     username = request.json.get("username")
244     if username is None:
245         abort(400) # missing args
246     account = g.account
247     other = Statement.findAccount(username)
248     if other is None:
249         abort(404)
250     success = Statement.removeFriends(account.id, other.id)
251     if success:
252         return jsonify(data=[]), 204
253     else:
254         abort(404)
255
256

```

```

257 @main.route("/lobby/friends")
258 @auth.login_required
259 def getFriendLobbies():
260     friends = Statement.getFriends(g.account.id)
261     lobbyInfo = lambda lobby: { # noqa: E731
262         "lobby-id": lobby.id,
263         "game-id": lobby.gameId,
264         "game-name": Statement.getGame(lobby.gameId).name,
265     }
266     accountInfo = lambda account: { # noqa: E731
267         "account-id": account.id,
268         "username": account.username,
269     }
270     lobbies = [
271         {
272             "account": accountInfo(account),
273             "lobbies": [
274                 lobbyInfo(lobby) for lobby in
275                 lobbyHandler.getMember(account.id)
276             ],
277             for account in friends
278             if len(lobbyHandler.getMember(account.id)) > 0
279         ]
280     return jsonify(lobbies)

```

```

1 # server.models
2 import datetime
3
4 import jwt
5 from flask import current_app
6 from flask_sqlalchemy import SQLAlchemy
7 from werkzeug.security import check_password_hash, generate_password_hash
8
9 import udp.auth as auth
10
11 db = SQLAlchemy()
12
13
14 # models
15 class Friends(db.Model):
16     account_one_id = db.Column(
17         db.Integer, db.ForeignKey("account.id"), primary_key=True

```



```

18     )
19     account_two_id = db.Column(
20         db.Integer, db.ForeignKey("account.id"), primary_key=True
21     )
22
23
24 class Scores(db.Model):
25     id = db.Column(db.Integer, primary_key=True)
26     score = db.Column(db.Integer, nullable=False)
27     account_id = db.Column(db.Integer, db.ForeignKey("account.id"),
28         nullable=False)
29     game_id = db.Column(db.Integer, db.ForeignKey("game.id"), nullable=False)
30
31 class Account(db.Model):
32     id = db.Column(db.Integer, primary_key=True)
33     username = db.Column(db.String(255), unique=True, nullable=False)
34     password = db.Column(db.String(162), nullable=False)
35     private_key = db.Column(db.LargeBinary(1337))
36     public_key = db.Column(db.LargeBinary(294))
37
38     def hashPassword(self, password: str) -> None:
39         self.password = generate_password_hash(password)
40
41     def verifyPassword(self, password: str) -> bool:
42         return check_password_hash(self.password, password)
43
44     def generateToken(self, expiration: int = 600) -> str:
45         data = {
46             "id": self.id,
47             "exp": datetime.datetime.now() +
48                 datetime.timedelta(seconds=expiration),
49         }
50         token = jwt.encode(data, current_app.config["SECRET_KEY"],
51             algorithm="HS256")
52         return token
53
54     @staticmethod
55     def validateToken(token: str):
56         try:
57             data = jwt.decode(
58                 token,
59                 current_app.config["SECRET_KEY"],
60                 leeway=datetime.timedelta(seconds=10),
61                 algorithms=["HS256"],

```

```

60         )
61         except: # noqa: E722
62             return None
63         account = Statement.getAccount(data.get("id"))
64         return account
65
66     def generateKey(self, password: bytes) -> None:
67         k = auth.generateRsaKey()
68         self.private_key = auth.getDerFromRsaPrivate(k, password)
69         self.public_key = auth.getDerFromRsaPublic(k.public_key())
70
71     @staticmethod
72     def decryptKey(self, key: bytes, password: bytes) ->
73         auth.rsa.RSAPublicKey:
74         k = auth.getRsaPrivateFromDer(key, password)
75         return k
76
77 class Game(db.Model):
78     id = db.Column(db.Integer, primary_key=True)
79     name = db.Column(db.String(255), unique=True, nullable=False)
80     min_players = db.Column(db.Integer, default=1)
81     max_players = db.Column(db.Integer)
82
83
84 class Statement:
85     # get
86     @staticmethod
87     def getGame(gameId: int) -> Game:
88         return Game.query.filter_by(id=gameId).scalar()
89
90     @staticmethod
91     def getGames() -> list[Game]:
92         return Game.query.all()
93
94     @staticmethod
95     def getAccount(userId: int) -> Account:
96         return Account.query.filter_by(id=userId).scalar()
97
98     @staticmethod
99     def getFriends(accountId: int) -> list[Account]:
100         friends = Friends.query.filter(
101             (Friends.account_one_id == accountId)
102             | (Friends.account_two_id == accountId)
103         )

```

```

104         friends = [
105             friend.account_one_id
106             if friend.account_one_id != accountId
107             else friend.account_two_id
108             for friend in friends
109         ]
110         friends = [Statement.getAccount(id) for id in friends]
111         return friends
112
113     # create
114     @staticmethod
115     def createAccount(username: str, password: str) -> Account:
116         account = Account(username=username)
117         account.hashPassword(password)
118         account.generateKey(password.encode())
119         db.session.add(account)
120         db.session.commit()
121         return account
122
123     @staticmethod
124     def createFriends(accountIdOne: int, accountIdTwo: int) -> Friends:
125         idOne = min(accountIdOne, accountIdTwo)
126         idTwo = max(accountIdOne, accountIdTwo)
127         friends = Friends(account_one_id=idOne, account_two_id=idTwo)
128         db.session.add(friends)
129         db.session.commit()
130         return friends
131
132     @staticmethod
133     def createGame(id:int, name:str, min_players:int, max_players:int) ->
134         Game:
135         game = Game(id=id, name=name, min_players=min_players,
136                     max_players=max_players)
137         db.session.add(game)
138         db.session.commit()
139         return game
140
141     # find
142     @staticmethod
143     def findAccount(username: str) -> Account | None:
144         return Account.query.filter_by(username=username).scalar()
145
146     @staticmethod
147     def validateToken(token: str) -> Account | None:
148         return Account.validateToken(token)

```

```

147
148     @staticmethod
149     def findGame(gameName: str) -> Game | None:
150         return Game.query.filter_by(name=gameName).scalar()
151
152     # delete
153     @staticmethod
154     def removeFriends(accountIdOne: int, accountIdTwo: int) -> bool:
155         idOne = min(accountIdOne, accountIdTwo)
156         idTwo = max(accountIdOne, accountIdTwo)
157         friends = Friends.query.filter(
158             (Friends.account_one_id == idOne) & (Friends.account_two_id ==
159                 idTwo)
160         )
161         if friends is not None:
162             friends.delete()
163             db.session.commit()
164             return True
165         else:
166             return False

```

9.3.3 rps

```

1 # rps.__init__
2 import os
3
4 import yaml
5
6
7 class bcolors:
8     HEADER = "\033[95m"
9     OKBLUE = "\033[94m"
10    OKCYAN = "\033[96m"
11    OKGREEN = "\033[92m"
12    WARNING = "\033[93m"
13    FAIL = "\033[91m"
14    ENDC = "\033[0m"
15    BOLD = "\033[1m"
16    UNDERLINE = "\033[4m"
17
18
19 class Choice:
20     ROCK = 0

```

```

21     PAPER = 1
22     SCISSORS = 2
23
24
25 class Outcome:
26     LOOSE = 0
27     WIN = 1
28     DRAW = 2
29
30
31 QUEUE_TIMEOUT = 10
32
33 # config
34 CONFIG_PATH = os.path.join(os.path.dirname(__file__), "game_config.yaml")
35
36 with open(CONFIG_PATH) as f:
37     config = yaml.safe_load(f)
38
39 ID = config["ID"]
40 NAME = config["NAME"]
41 MIN_PLAYERS = config["MIN_PLAYERS"]
42 MAX_PLAYERS = config["MAX_PLAYERS"]

```

```

1 # rps.__main__
2 import threading
3
4 from . import client, server
5
6
7 def runServer():
8     s = server.Server((S_HOST, S_PORT))
9     sT = threading.Thread(target=s.mainloop, daemon=True)
10    sT.start()
11    return s, sT
12
13
14 def runClient():
15     c = client.Client((C_HOST, C_PORT), (S_HOST, S_PORT))
16     return c
17
18
19 if __name__ == "__main__":
20     import time

```

```

21
22     from udp import C_HOST, C_PORT, S_HOST, S_PORT
23
24     print("\n" * 4)
25     s, sT = runServer()
26     time.sleep(1)
27     c = runClient()
28     c.connect()
29     time.sleep(1)
30     c.isRunning = False
31     time.sleep(1)
32     s.isRunning = False
33     time.sleep(1)
34     print("END")

```

```

1  # rps.client
2  import json
3  from queue import Empty, Queue
4  from threading import Thread
5
6  import udp.error as error
7  from inputtimeout import TimeoutOccurred, inputtimeout
8  from udp.auth import rsa
9  from udp.client import Client as UdpClient
10 from udp.packet import Flag, lazyFlags
11
12 from . import QUEUE_TIMEOUT, Outcome, bcolors
13
14
15 class Client:
16     isRunning: bool
17     recvQueue: Queue
18     score: int
19     onReceiveData: None
20     gameThread: Thread
21     udpClient: UdpClient
22
23     def __init__(
24         self,
25         addr: tuple[str, int],
26         targetAddr: tuple[str, int],
27         rsaKey: rsa.RSAPrivateKey|None = None,
28         userId: int | str | None = None,

```

```

29     username: str | None = None,
30     onReceiveData=None,
31 ) -> None:
32     self.isRunning = True
33     self.recvQueue = Queue()
34     self.score = 0
35     self.onReceiveData = onReceiveData
36     self.gameThread = Thread(
37         name=f"{addr[1]}:Gameloop", target=self.gameloop, daemon=True
38     )
39     self.udpClient = UdpClient(
40         addr,
41         targetAddr,
42         rsaKey=rsaKey,
43         accountId=userId,
44         username=username,
45         onConnect=self.onConnect,
46         onReceiveData=self.receive,
47     )
48
49     def send(self, addr: tuple[str, int], data: json) -> None:
50         self.udpClient.queueDefault(
51             addr, flags=lazyFlags(Flag.RELIABLE), data=self.encodeData(data)
52         )
53
54     def receive(self, addr: tuple[str, int], data: bytes):
55         self.recvQueue.put((addr, self.decodeData(data)))
56         if self.onReceiveData:
57             self.onReceiveData(addr, data)
58
59     @staticmethod
60     def encodeData(data: dict) -> bytes:
61         return json.dumps(data).encode()
62
63     @staticmethod
64     def decodeData(data: bytes) -> dict:
65         return json.loads(data.decode())
66
67     def connect(self) -> None:
68         try:
69             self.udpClient.connect()
70         except error.PaperClipError as e:
71             match e:
72                 case error.NoSpaceError():
73                     print(

```

```

74         f"{bcolors.FAIL}Failed to join server due to
           {error.ConnectionErrorCodes.NO_SPACE.name}:
           {e.args[0]}{bcolors.ENDC}"
75     )
76     case error.CertificateInvalidError():
77         print(
78             f"{bcolors.FAIL}Failed to join server due to
              {error.ConnectionErrorCodes.CERTIFICATE_INVALID.name}:
              {e.args[0]}{bcolors.ENDC}"
79         )
80     case error.FinishInvalidError():
81         print(
82             f"{bcolors.FAIL}Failed to join server due to
              {error.ConnectionErrorCodes.FINISH_INVALID.name}:
              {e.args[0]}{bcolors.ENDC}"
83         )
84     case _:
85         raise e
86
87 def onConnect(self, addr: tuple[str, int]) -> None:
88     self.gameThread.start()
89     try:
90         self.udpClient.mainloop(self.quit)
91     except error.PaperClipError as e:
92         match e:
93             case error.ServerDisconnectError():
94                 print(
95                     f"{bcolors.FAIL}Server connection terminated due to
                      {error.DisconnectErrorCodes.SERVER_DISCONNECT.name}:
                      {e.args[0]}\nPlease wait while connection closes
                      gracefully...{bcolors.ENDC}"
96                 )
97             case _:
98                 raise e
99     if self.gameThread.is_alive():
100         self.gameThread.join()
101     return None
102
103 def gameloop(self) -> None:
104     print(f"{bcolors.HEADER}\n\nRock Paper Scissors{bcolors.ENDC}")
105     try:
106         while self.isRunning:
107             choice = None
108             print("Choice R[0], P[1], S[2]: ")
109             while choice is None:

```



```

110         try:
111             choice = inputtimeout("", timeout=10).strip()
112             if choice == "q":
113                 print(
114                     f"{bcolors.FAIL}Quitting. Please
115                     wait...{bcolors.ENDC}"
116                 )
117                 self.isRunning = False
118                 break
119             choice = int(choice)
120             if choice not in (0, 1, 2):
121                 print(
122                     f"{bcolors.FAIL}Invalid choice
123                     '{choice}'.{bcolors.ENDC}"
124                 )
125                 choice = None
126             except ValueError:
127                 print(f"{bcolors.FAIL}Invalid choice.{bcolors.ENDC}")
128                 choice = None
129             except KeyboardInterrupt:
130                 print(f"{bcolors.FAIL}Quitting. Please
131                 wait...{bcolors.ENDC}")
132                 self.isRunning = False
133                 break
134             except TimeoutOccurred:
135                 if not self.isRunning:
136                     break
137         if self.isRunning:
138             self.send(self.udpClient.targetAddr, {"choice": choice})
139             print("Waiting for other player...")
140             while self.isRunning:
141                 try:
142                     addr, data =
143                         self.recvQueue.get(timeout=QUEUE_TIMEOUT)
144                     break
145                 except Empty:
146                     pass # check still running
147             if self.isRunning:
148                 match data["outcome"]:
149                     case 0:
150                         o = f"You {bcolors.FAIL}LOOSE{bcolors.ENDC}. "
151                     case 1:
152                         o = f"You {bcolors.OKGREEN}WIN{bcolors.ENDC}.
153                         "
154                     case 2:

```

```

150         o = f"You {bcolors.OKCYAN}DRAW{bcolors.ENDC}."
151         "
152         case _:
153             o = ""
154         print(
155             f"\n{o}You Picked {data['choice']}. They picked
156             {data['otherChoice']}. \nThe score is
157             {data['score']['score']}: {data['otherScore']['score']}."
158         )
159         if data["outcome"] == Outcome.WIN:
160             self.score += 1
161             self.recvQueue.task_done()
162     finally:
163         self.udpClient._quit()
164
165     def quit(self, msg: str = "quit call", e: Exception | None = None) ->
166         None:
167         self.isRunning = False
168         self.udpClient.quit(msg, e)

```

```

1 # game_config.yaml
2 NAME: "RPS"
3 ID: 1
4 MIN_PLAYERS: 2
5 MAX_PLAYERS: 2

```

```

1 # rps.server
2 import json
3 from queue import Empty, Queue
4 from threading import Lock
5
6 from udp.auth import rsa
7 from udp.packet import Flag, lazyFlags
8 from udp.server import Server as UdpServer
9
10 from . import MAX_PLAYERS, QUEUE_TIMEOUT, Choice, Outcome
11
12
13 class Server:
14     isRunning: bool
15     recvBuffer: Queue

```

```

16  players: dict[tuple[str, int], dict[str, int]]
17  playersLock: Lock
18  udpServer: UdpServer
19  onClientJoin: None
20  onClientLeave: None
21  onReceiveData: None
22
23  def __init__(
24      self,
25      addr: tuple[str, int],
26      rsaKey: rsa.RSAPrivateKey | None = None,
27      onClientJoin=None,
28      onClientLeave=None,
29      onReceiveData=None,
30  ):
31      self.isRunning = True
32      self.recvQueue = Queue()
33      self.players = {}
34      self.playersLock = Lock()
35      self.onClientJoin = onClientJoin
36      self.onClientLeave = onClientLeave
37      self.onReceiveData = onReceiveData
38      self.udpServer = UdpServer(
39          addr,
40          maxClients=MAX_PLAYERS,
41          rsaKey=rsaKey,
42          onClientJoin=self.playerJoin,
43          onClientLeave=self.playerLeave,
44          onReceiveData=self.receive,
45      )
46
47  def send(self, addr: tuple[str, int], data: dict) -> None:
48      self.udpServer.queueDefault(
49          addr, flags=lazyFlags(flag.RELIABLE), data=self.encodeData(data)
50      )
51
52  def receive(self, addr: tuple[str, int], data: bytes) -> None:
53      self.recvQueue.put((addr, self.decodeData(data)))
54      if self.onReceiveData:
55          self.onReceiveData(addr, data)
56
57  @staticmethod
58  def encodeData(data: dict) -> bytes:
59      return json.dumps(data).encode()
60

```

```

61 @staticmethod
62 def decodeData(data: bytes) -> dict:
63     return json.loads(data.decode())
64
65 @staticmethod
66 def evaluateWin(choiceOne: int, choiceTwo: int) -> int:
67     match choiceOne:
68         case Choice.ROCK:
69             match choiceTwo:
70                 case Choice.ROCK:
71                     return Outcome.DRAW
72                 case Choice.PAPER:
73                     return Outcome.LOOSE
74                 case Choice.SCISSORS:
75                     return Outcome.WIN
76                 case _:
77                     raise ValueError
78         case Choice.PAPER:
79             match choiceTwo:
80                 case Choice.ROCK:
81                     return Outcome.WIN
82                 case Choice.PAPER:
83                     return Outcome.DRAW
84                 case Choice.SCISSORS:
85                     return Outcome.LOOSE
86                 case _:
87                     raise ValueError
88         case Choice.SCISSORS:
89             match choiceTwo:
90                 case Choice.ROCK:
91                     return Outcome.LOOSE
92                 case Choice.PAPER:
93                     return Outcome.WIN
94                 case Choice.SCISSORS:
95                     return Outcome.DRAW
96                 case _:
97                     raise ValueError
98     case _:
99         raise ValueError
100
101 @staticmethod
102 def evaluatePlayerChoices(choices: list[tuple[tuple[str, int], int]]):
103     outcomes = [
104         (choices[0][0], Server.evaluateWin(choices[0][1], choices[1][1])),
105         (choices[1][0], Server.evaluateWin(choices[1][1], choices[0][1])),

```

```

106     ]
107     return outcomes
108
109 def getChoices(self) -> list[tuple[tuple[str, int], int]]:
110     choices = {}
111     while self.isRunning:
112         try:
113             addr, data = self.recvQueue.get(timeout=QUEUE_TIMEOUT)
114             choices[addr] = data["choice"]
115             if len(choices) == 2:
116                 choices = [(addr, choice) for addr, choice in
                             choices.items()]
117                 self.recvQueue.task_done()
118                 return choices
119         except Empty:
120             pass # check still running
121
122 def playerJoin(self, addr: tuple[str, int], accountId: int) -> None:
123     with self.playersLock:
124         self.players[addr] = {"score": 0, "accountId": accountId}
125     if self.onClientJoin:
126         self.onClientJoin(addr, accountId)
127
128 def playerLeave(self, addr: tuple[str, int], accountId: int) -> None:
129     with self.playersLock:
130         # TODO: submit score
131         del self.players[addr]
132     if self.onClientLeave:
133         self.onClientLeave(addr, accountId)
134
135 def isNotFull(self) -> bool:
136     return self.udpServer.isNotFull()
137
138 def isEmpty(self) -> bool:
139     return self.udpServer.isEmpty()
140
141 def getPlayers(self) -> dict[tuple[str, int], dict[str, int]]:
142     with self.playersLock:
143         return self.players.copy()
144
145 def getPlayer(self, addr: tuple[str, int]) -> int:
146     with self.playersLock:
147         if addr in self.players:
148             return self.players[addr]
149         else:

```

```

150         return None
151
152     def setPlayer(self, addr: tuple[str, int], v: int) -> None:
153         with self.playersLock:
154             if addr in self.players:
155                 self.players[addr] = v
156
157     def incrementPlayer(self, addr: tuple[str, int]) -> None:
158         with self.playersLock:
159             self.players[addr]["score"] += 1
160
161     def getAccountId(self, addr: tuple[str, int]) -> int:
162         with self.playersLock:
163             return self.players[addr]["accountId"]
164
165     def getAccountIds(self, addr: tuple[str, int]) -> list[int]:
166         with self.playersLock:
167             return [player["accountId"] for player in self.players.values()]
168
169     @property
170     def playerCount(self) -> int:
171         with self.playersLock:
172             return len(self.players)
173
174     def mainloop(self) -> None:
175         self.udpServer.startThreads()
176         try:
177             while self.isRunning:
178                 if self.playerCount == MAX_PLAYERS:
179                     choices = self.getChoices()
180                     outcomes = self.evaluatePlayerChoices(choices)
181                     replies = {}
182                     for addr, outcome in outcomes:
183                         replies[addr] = {
184                             "outcome": outcome,
185                             "choice": [v for k, v in choices if k == addr][0],
186                             "otherChoice": [v for k, v in choices if k !=
187                                             addr][0],
188                         }
189                     if outcome == Outcome.WIN:
190                         self.incrementPlayer(addr)
191                     scores = self.getPlayers()
192                     for addr in replies:
193                         replies[addr] |= {
194                             "score": scores[addr],

```

```

194         "otherScore": [v for k, v in scores.items() if k
195                         != addr][
196                             0
197                         ],
198     }
199     self.send(addr, replies[addr])
200
201     finally:
202         self.quit()
203
204     def quit(self) -> None:
205         self.isRunning = False
206         self.udpServer.quit()

```

9.3.4 client

```

1 # client.__init__
2 import os
3 import sys
4
5 import dotenv
6
7 from udp import logger, logging
8
9 dotenv.load_dotenv()
10
11 TCP_HOST = os.environ.get("S_HOST")
12 TCP_PORT = int(os.environ.get("TCP_PORT"))
13 C_PORT = int(os.environ.get("C_PORT"))
14 SERVER_URL = f"http://{TCP_HOST}:{TCP_PORT}"
15
16 offset = sys.argv[1:]
17 try:
18     offset = int(offset[0])
19 except ValueError:
20     offset = None
21 except IndexError:
22     offset = None
23
24 if os.environ.get("DEBUG") is not None:
25     logger.setLevel(logging.WARNING)
26     while offset is None:
27         try:
28             offset = int(input("\noffset: "))

```

```

29         except ValueError:
30             print("Invalid input.")
31     else:
32         logger.setLevel(logging.ERROR)
33
34     if offset is not None:
35         C_PORT += offset

```

```

1  # client.__main__
2  import base64
3  import json
4  import time
5
6  import requests
7  from requests.auth import HTTPBasicAuth
8
9  import udp.auth
10 from rps.client import Client as RpsClient
11 from udp import bcolors
12
13 from . import C_PORT, SERVER_URL, TCP_HOST
14
15
16 class Client:
17     id: int
18     username: str
19     password: str
20     gameClient: None
21     token: str
22     key: udp.auth.rsa.RSAPublicKey
23     auth: HTTPBasicAuth
24
25     def __init__(self, username: str, password: str, token: str | None =
        None) -> None:
26         self.username = username
27         self.password = password
28         self.gameClient = None
29         self.token = (
30             token if token is not None else self.getToken(self.username,
                self.password)
31         )
32         self.auth = HTTPBasicAuth(self.token, "")
33         self.getKey(password.encode())

```



```

34
35 # auth
36 @staticmethod
37 def getToken(username: str, password: str) -> str:
38     url = SERVER_URL + "/auth/token"
39     r = requests.get(url, auth=(username, password))
40     assert r.status_code == 200, r
41     return r.json()["token"]
42
43 @staticmethod
44 def createAccount(username: str, password: str) -> str:
45     url = SERVER_URL + "/auth/register"
46     headers = {"Content-Type": "application/json"}
47     data = {"username": username, "password": password}
48     r = requests.post(url, headers=headers, data=json.dumps(data))
49     assert r.status_code == 201, r
50     return r.json()["username"]
51
52 def getKey(self, password: bytes) -> udp.auth.rsa.RSAPrivateKey:
53     url = SERVER_URL + "/auth/key"
54     r = requests.get(url, auth=self.auth)
55     assert r.status_code == 200, r
56     self.id = r.json()["account-id"]
57     key = base64.decodebytes(r.json()["key"].encode())
58     self.key = udp.auth.getRsaPrivateFromDer(key, password)
59
60 # game
61 def getGames(self) -> dict:
62     url = SERVER_URL + "/games/"
63     r = requests.get(url, auth=self.auth)
64     assert r.status_code == 200, r
65     return r.json()
66
67 def getLobbies(self) -> dict:
68     url = SERVER_URL + "/lobby/all"
69     r = requests.get(url, auth=self.auth)
70     assert r.status_code == 200, r
71     return r.json()
72
73 def createLobby(
74     self, gameId: int | None = None, gameName: str | None = None
75 ) -> dict:
76     url = SERVER_URL + "/lobby/create"
77     headers = {"Content-Type": "application/json"}
78     data = {}

```

```

79         if gameId:
80             data["game-id"] = gameId
81         elif gameName:
82             data["game-name"] = gameName
83         r = requests.post(url, headers=headers, data=json.dumps(data),
84                           auth=self.auth)
85         assert r.status_code == 201, r
86         return r.json()
87
88     def getLobby(self, lobbyId: int) -> dict:
89         url = SERVER_URL + "/lobby/"
90         headers = {"Content-Type": "application/json"}
91         data = {"lobby-id": lobbyId}
92         r = requests.get(url, headers=headers, data=json.dumps(data),
93                           auth=self.auth)
94         assert r.status_code == 200
95         return r.json()
96
97     def findLobby(self, gameId: int | None = None, gameName: str | None =
98         None) -> dict:
99         url = SERVER_URL + "/lobby/find"
100        headers = {"Content-Type": "application/json"}
101        data = {}
102        if gameId:
103            data["game-id"] = gameId
104        elif gameName:
105            data["game-name"] = gameName
106        r = requests.get(url, headers=headers, data=json.dumps(data),
107                          auth=self.auth)
108        assert r.status_code == 200, r
109        return r.json()
110
111    # friends
112    def friendLobbies(self) -> dict:
113        url = SERVER_URL + "/lobby/friends"
114        r = requests.get(url, auth=self.auth)
115        assert r.status_code == 200, r
116        return r.json()
117
118    def getFriends(self) -> dict:
119        url = SERVER_URL + "/friends/"
120        r = requests.get(url, auth=self.auth)
121        assert r.status_code == 200, r
122        return r.json()

```

```

120 def addFriend(self, username: str) -> dict:
121     url = SERVER_URL + "/friends/add"
122     headers = {"Content-Type": "application/json"}
123     data = {"username": username}
124     r = requests.post(url, headers=headers, data=json.dumps(data),
125                       auth=self.auth)
126     assert r.status_code == 201, r
127     return r.json()
128
129 def removeFriend(self, username: str) -> bool:
130     url = SERVER_URL + "/friend/remove"
131     headers = {"Content-Type": "application/json"}
132     data = {"username": username}
133     r = requests.delete(url, headers=headers, data=json.dumps(data),
134                       auth=self.auth)
135     assert r.status_code == 204, r
136     return True
137
138 # join
139 def join(self, lobbyId: int) -> None:
140     print(f"\n{bcolors.WARNING}Joining Lobby '{lobbyId}'{bcolors.ENDC}")
141     data = self.getLobby(lobbyId)
142     if data["lobby-addr"] is not None:
143         match data["game-id"]:
144             case 1:
145                 self.gameClient = RpsClient(
146                     (TCP_HOST, C_PORT),
147                     data["lobby-addr"],
148                     rsaKey=self.key,
149                     userId=self.id,
150                     username=self.username,
151                 )
152                 self.gameClient.connect()
153             case _:
154                 raise ValueError(f"Unknown gameId {data['game-id']}")
155
156 def mainloop():
157     print(f"{bcolors.HEADER}\nLobby.{bcolors.ENDC}")
158     print("1. Login\n2. Register\n3. Quit")
159     while True:
160         option = input(": ").strip()
161         match option:
162             case "1":
163                 _login()

```

```

163         break
164     case "2":
165         _register()
166         break
167     case "3":
168         break
169     case _:
170         print(f"{bcolors.FAIL}Error: Invalid input
171             '{option}'.{bcolors.ENDC}")
172
173 def _register(username: str | None = None, password: str | None = None):
174     print(f"{bcolors.HEADER}\nRegister.{bcolors.ENDC}")
175     account = None
176     while account is None:
177         while username is None or password is None:
178             try:
179                 username = input("Username: ").strip()
180                 password = input("Password: ").strip()
181             except: # noqa: E722
182                 print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")
183         try:
184             account = Client.createAccount(username, password)
185         except AssertionError:
186             print(
187                 f"{bcolors.FAIL}Account could not be created. Please try
188                 again.{bcolors.ENDC}\n"
189             )
190             username = None
191             password = None
192     else:
193         print(f"Account Created for '{account}'")
194         _login(username, password)
195
196 def _login(username: str | None = None, password: str | None = None):
197     print(f"{bcolors.HEADER}\nLogin.{bcolors.ENDC}")
198     token = None
199     while token is None:
200         while username is None or password is None:
201             try:
202                 username = input("Username: ").strip()
203                 password = input("Password: ").strip()
204             except: # noqa: E722
205                 print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")

```

```

206     try:
207         token = Client.getToken(username, password)
208     except AssertionError:
209         print(
210             f"{bcolors.FAIL}Invalid login details. Please try
                again.{bcolors.ENDC}\n"
211         )
212         username = None
213         password = None
214     else:
215         client = Client(username, password, token)
216         _menu(client)
217
218
219 def _menu(client):
220     isRunning = True
221     while isRunning:
222         print(f"\n{bcolors.HEADER}Main Menu{bcolors.ENDC}")
223         print(f"{bcolors.OKGREEN>Hello {client.username}.{bcolors.ENDC}")
224         while True:
225             print(
226                 "\n1. Manage friends\n2. See available games\n3. Start or
                    join a lobby\n4. Quit"
227             )
228             option = input(": ").strip()
229             match option:
230                 case "1":
231                     _friends(client)
232                     break
233                 case "2":
234                     _game(client)
235                     break
236                 case "3":
237                     _lobby(client)
238                     break
239                 case "4":
240                     isRunning = False
241                     break
242                 case _:
243                     print(
244                         f"{bcolors.FAIL}Error: Invalid input
                            '{option}'.{bcolors.ENDC}"
245                     )
246
247

```

```

248 def _friends(client: Client):
249     while True:
250         print(f"{bcolors.HEADER}\nFriends.{bcolors.ENDC}")
251         friends = client.getFriends()
252         friends = "\n\t".join(
253             [
254                 f"{i+1}. {friend['username']}"
255                 for i, friend in enumerate(friends["friends"])
256             ]
257         )
258         print(f"Friend list: \n\t{friends}")
259         print("\n1. Add New Friend\n2. Remove Friend\n3. Return to Main Menu")
260         while True:
261             option = input(": ").strip()
262             match option:
263                 case "1" | "2":
264                     username = input("\nUsername: ").strip()
265                     match option:
266                         case "1":
267                             try:
268                                 client.addFriend(username)
269                                 print(
270                                     f"\n{bcolors.OKGREEN}Account '{username}'
                                         added as friend{bcolors.ENDC}"
271                                 )
272                                 break
273                             except AssertionError:
274                                 print(
275                                     f"\n{bcolors.FAIL}Error: No such account
                                         with username
                                         '{username}'.{bcolors.ENDC}"
276                                 )
277                         case "2":
278                             try:
279                                 client.removeFriend(username)
280                                 print(
281                                     f"\n{bcolors.OKGREEN}Account '{username}'
                                         removed as friend{bcolors.ENDC}"
282                                 )
283                                 break
284                             except AssertionError:
285                                 print(
286                                     f"\n{bcolors.FAIL}Error: No such account
                                         with username '{username}' in friend
                                         list.{bcolors.ENDC}"

```

```

287         )
288         case "3":
289             return None
290         case _:
291             print(
292                 f"{bcolors.FAIL}Error: Invalid input
293                 '{option}'.{bcolors.ENDC}"
294             )
295
296 def _game(client: Client):
297     while True:
298         print(f"{bcolors.HEADER}\nGames{bcolors.ENDC}")
299         availableGames = client.getGames()
300         availableGames = "\n\t".join(
301             [f"{id}. {game}" for id, game in availableGames.items()]
302         )
303         print(f"Available Games: \n\t{availableGames}")
304         input("\nPress enter to return to main menu: ")
305         return None
306
307
308 def _lobby(client: Client):
309     while True:
310         print(f"{bcolors.HEADER}\nLobby.{bcolors.ENDC}")
311         print(
312             "\n1. Matchmaking\n2. See Friends' Lobbies\n3. Join Lobby\n4.
313             Create Lobby\n5. Return to Main Menu"
314         )
315         while True:
316             option = input(": ").strip()
317             match option:
318                 case "1":
319                     _matchmaking(client)
320                     break
321                 case "2":
322                     _friendsLobbies(client)
323                     break
324                 case "3":
325                     _joinLobby(client)
326                     break
327                 case "4":
328                     _createLobby(client)
329                     break
330                 case "5":

```

```

330         return None
331     case _:
332         print(
333             f"{bcolors.FAIL}Error: Invalid input
334             '{option}'.{bcolors.ENDC}"
335         )
336
337 def _matchmaking(client: Client):
338     print(f"{bcolors.HEADER}\nMatchmaking.{bcolors.ENDC}")
339     game = _gameInput(client)
340     if game is None:
341         return None
342     try:
343         lobby = client.findLobby(gameName=game)
344     except AssertionError:
345         lobby = client.createLobby(gameName=game)
346     time.sleep(1)
347     client.join(lobby["lobby-id"])
348     return None
349
350
351 def _friendsLobbies(client: Client):
352     print(f"{bcolors.HEADER}\nFriends' Lobbies.{bcolors.ENDC}")
353     lobbies = client.friendLobbies()
354     lobbiesInfo = lambda lobbies: "\n\t\t".join(
355         [
356             f"{bcolors.OKCYAN}{lobby['lobby-id']}{bcolors.ENDC}.
357             {lobby['game-name']}"
358         ]
359     ) # noqa: E731
360     lobbies = "\n\t\t".join(
361         [
362             f"\n\t\t{i+1}.
363             {account['account']['username']}: \n\t\t\t{lobbiesInfo(account['lobbies'])}"
364             for i, account in enumerate(lobbies)
365         ]
366     )
367     print(f"\nLobbies:{lobbies}")
368     print(
369         f"Input {bcolors.OKCYAN}Lobby Id{bcolors.ENDC} to Join Friend or
370         Press Enter to Return to Menu."
371     )
372     while True:

```



```

371     option = input(": ").strip()
372     if option == "":
373         return None
374     else:
375         try:
376             option = int(option)
377             client.join(option)
378             return None
379         except ValueError:
380             print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")
381
382
383 def _joinLobby(client: Client):
384     print(f"{bcolors.HEADER}\nJoin Lobby.{bcolors.ENDC}")
385     lobbyId = None
386     while lobbyId is None:
387         try:
388             lobbyId = input("\nLobby Id: ").strip()
389             if lobbyId == "":
390                 return None
391             else:
392                 lobbyId = int(lobbyId)
393         except ValueError:
394             print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")
395     try:
396         client.join(lobbyId)
397         return None
398     except: # noqa: E722
399         return None
400
401
402 def _createLobby(client: Client):
403     print(f"{bcolors.HEADER}\nCreate Lobby.{bcolors.ENDC}")
404     game = _gameInput(client)
405     while True:
406         if game is None:
407             return None
408         try:
409             lobby = client.createLobby(gameName=game)
410             client.join(lobby["lobby-id"])
411             return None
412         except AssertionError:
413             print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")
414
415

```

```

416 def _gameInput(client: Client) -> str:
417     availableGames = client.getGames()
418     games = "\n\t".join([f"{id}. {game}" for id, game in
        availableGames.items()])
419     print(f"\nAvailable Games: \n\t{games}")
420     game = None
421     while game is None or game.lower() not in map(
422         lambda x: x.lower(), availableGames.values()
423     ):
424         try:
425             game = input("Game: ").strip()
426             if game == "":
427                 return None
428             except: # noqa: E722
429                 print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")
430     return game
431
432
433 if __name__ == "__main__":
434     mainloop()

```

```

1 # .env
2 # udp
3 S_HOST=127.0.0.1
4 S_PORT=2024
5 C_HOST=127.0.0.1
6 C_PORT=2025
7 ## node
8 SOCKET_BUFFER_SIZE = 1024
9 SEND_SLEEP_TIME = 0.1
10 QUEUE_TIMEOUT = 10
11 SOCKET_TIMEOUT = 20
12 ## server
13 HEARTBEAT_MAX_TIME = 120
14 HEARTBEAT_MIN_TIME = 30
15 MAX_CLIENTS
16 ## auth
17 ORG_NAME = Paperclip
18 COMMON_NAME = 127.0.0.1
19 ## utils
20 MAX_FRAGMENT_SIZE = 988
21
22 # client

```

```

23 TCP_PORT = 5000
24
25 # app
26 FLASK_APP = server
27 PRUNE_TIME = 58
28 SECRET_KEY = MyVerySecretKey
29 SQLALCHEMY_DATABASE_URI = mysql://root:root@localhost:3306/paperclip
30
31 # debug
32 DEBUG = True

```

```

1 cryptography==42.0.5
2 Flask==3.0.2
3 Flask-HTTPAuth==4.8.0
4 Flask-SQLAlchemy==3.1.1
5 SQLAlchemy-Utills==0.41.2
6 mysqlclient==2.2.4
7 requests==2.31.0
8 PyJWT==2.8.0
9 pytest==8.1.1
10 python-dotenv==1.0.1
11 PyYAML==6.0.1

```

```

1 # test_udp
2 import os
3 import threading
4 from random import choice, randint
5
6 from udp import C_HOST, C_PORT, auth, error, node, packet, utils
7
8
9 ## node
10 def testNodeSequenceIdLock():
11     n = node.Node((C_HOST, C_PORT))
12
13     def test():
14         for _ in range(100000):
15             n.incrementSequenceId(n.addr)
16
17     threads = [threading.Thread(target=test) for _ in range(10)]
18     for t in threads:

```

```

19         t.start()
20     for t in threads:
21         t.join()
22     assert n.sequenceId == 16960, n.sequenceId
23
24
25 # error
26 def testErrorCode():
27     major = choice([i for i in error.Major])
28     minor = error.getMinor(major, randint(0, 2))
29     mm = (major, minor)
30     e = error.getError(*mm)()
31     c = error.getErrorCode(e.__class__)
32     assert mm == c, (mm, e, c)
33
34
35 def testErrorPacket():
36     h = genRandAttr(packet.Type.ERROR)
37     p = packet.ErrorPacket(*h)
38     p.data = b"This is a test error"
39     p.major = randint(1, 3)
40     match p.major:
41         case error.Major.CONNECTION:
42             p.minor = randint(0, 3)
43         case error.Major.DISCONNECT:
44             p.minor = randint(0, 2)
45         case error.Major.PACKET:
46             p.minor = randint(0, 9)
47         case _:
48             p.minor = 0
49     eP = p.pack(p)
50     dP = packet.unpack(eP)
51     assert p == dP, (p, eP, dP)
52
53
54 # Heartbeat
55 def testHeartbeatPacket():
56     h = genRandAttr(packet.Type.HEARTBEAT)
57     p = packet.HeartbeatPacket(*h)
58     p.heartbeat = True
59     eP = p.pack(p)
60     dP = packet.unpack(eP)
61     assert p == dP, (p, eP, dP)
62
63

```

```

64 # frag
65 def testDefrag():
66     h = genRandAttr()
67     data = os.urandom(16)
68     p = packet.Packet(*h)
69     p.flags[packet.Flag.FRAG.value] = 0
70     p.fragment_id = None
71     p.fragment_number = None
72     p.data = data
73     fP = p.fragment()
74     dP = fP[0].defragment(fP)
75     assert p == dP, (p, fP, dP)
76
77
78 ## utils
79 def testDataCompress(d=os.urandom(16)):
80     cD = utils.compressData(d)
81     dD = utils.decompressData(cD)
82     assert d == dD, (d, cD, dD)
83
84
85 ## encrypt
86 def testPacketEncryption():
87     h = genRandAttr()
88     p = packet.Packet(*h)
89     p.flags[packet.Flag.ENCRYPTED.value] = 1
90     d = b"Hello World"
91     p.data = d
92     localKey = auth.generateEcKey()
93     peerKey = auth.generateEcKey()
94     localSessionKey = auth.generateSessionKey(localKey, peerKey.public_key())
95     peerSessionKey = auth.generateSessionKey(peerKey, localKey.public_key())
96     p.encryptData(localSessionKey)
97     # print(p.data)
98     p.decryptData(peerSessionKey)
99     # print(p.data)
100     assert d == p.data, (d, p.data)
101
102
103 ## auth
104 def sessionKey():
105     localKey = auth.generateEcKey()
106     peerKey = auth.generateEcKey()
107     localSessionKey = auth.generateSessionKey(localKey, peerKey.public_key())
108     peerSessionKey = auth.generateSessionKey(peerKey, localKey.public_key())

```

```

109     assert localSessionKey == peerSessionKey
110
111
112 def encryptDecrypt(inputText=b"Hello World"):
113     localKey = auth.generateEcKey()
114     peerKey = auth.generateEcKey()
115     sessionKey = auth.generateSessionKey(localKey, peerKey.public_key())
116     #
117     localCipher, iv = auth.generateCipher(sessionKey)
118     cipherText = auth.encryptBytes(localCipher, inputText)
119     #
120     peerCipher, _ = auth.generateCipher(sessionKey, iv)
121     outputText = auth.decryptBytes(peerCipher, cipherText)
122     #
123     assert inputText == outputText, (inputText, outputText)
124
125
126 ## packet
127 def genRandAttr(t=packet.Type.DEFAULT):
128     v, pT, sId = randint(0, 1), t, randint(0, 2**packet.SEQUENCE_ID_SIZE - 1)
129     f = [0 for _ in range(packet.FLAGS_SIZE)]
130     if randint(0, 1):
131         f[packet.Flag.FRAG.value] = 1
132         fId, fNum = (
133             randint(0, 2**packet.FRAGMENT_ID_SIZE - 1),
134             randint(0, 2**packet.FRAGMENT_NUM_SIZE - 1),
135         )
136     else:
137         fId, fNum = None, None
138     if randint(0, 1):
139         f[packet.Flag.ENCRYPTED.value] = 1
140         # iv = randint(0, 2**INIT_VECTOR_SIZE-1)
141         iv = auth.generateInitVector()
142     else:
143         iv = None
144     if randint(0, 1):
145         f[packet.Flag.CHECKSUM.value] = 1
146         c = randint(0, 2**packet.CHECKSUM_SIZE - 1)
147     else:
148         c = None
149     h = (v, pT, f, sId, fId, fNum, iv, c)
150     return h
151
152
153 def testAuth():

```

```

154     pK, c = (
155         auth.generateEcKey().public_key(),
156         auth.generateUserCertificate(auth.generateRsaKey()),
157     )
158     pKS, cS = (
159         packet.AuthPacket.getPublicKeyBytesSize(pK),
160         packet.AuthPacket.getCertificateByteSize(c),
161     )
162     h = (*genRandAttr(packet.Type.AUTH), pKS, pK, cS, c)
163     # static test
164     eH = packet.AuthPacket.encodeHeader(*h)
165     dH = packet.AuthPacket.decodeHeader(eH)[-1]
166     assert h == dH, (h, eH, dH)
167     # class tests
168     p = packet.AuthPacket(*h)
169     eP = p.pack(p)
170     dP = p.unpack(eP)
171     assert p == dP, (p, eP, dP)
172
173
174 def testAck():
175     # header
176     aId, aB = (
177         randint(0, 2**packet.ACK_ID_SIZE - 1),
178         [randint(0, 1) for _ in range(packet.ACK_BITS_SIZE)],
179     )
180     h = (*genRandAttr(packet.Type.ACK), aId, aB)
181     # static test
182     eH = packet.AckPacket.encodeHeader(*h)
183     dH = packet.AckPacket.decodeHeader(eH)[-1]
184     assert h == dH, (h, eH, dH)
185     # class tests
186     p = packet.AckPacket(*h)
187     eP = p.pack(p)
188     dP = p.unpack(eP)
189     assert p == dP, (p, eP, dP)
190
191
192 def testAckBits():
193     aId, aB = (
194         randint(0, 2**packet.ACK_ID_SIZE - 1),
195         [randint(0, 1) for _ in range(packet.ACK_BITS_SIZE)],
196     )
197     eAId, eAB = packet.AckPacket.encodeAckId(aId),
198         packet.AckPacket.encodeAckBits(aB)

```

```

198     dAId, dAB = packet.AckPacket.decodeAckId(eAId),
        packet.AckPacket.decodeAckBits(eAB)
199     assert (aId, aB) == (dAId, dAB), ((aId, aB), (eAId, eAB), (dAId, dAB))
200
201
202 def testDefault():
203     # header
204     h = genRandAttr()
205     # static test
206     eH = packet.Packet.encodeHeader(*h)
207     dH = packet.Packet.decodeHeader(eH)[: -1]
208     assert h == dH, (h, eH, dH)
209     # class tests
210     p = packet.Packet(*h)
211     eP = p.pack(p)
212     dP = p.unpack(eP)
213     assert p == dP, (p, eP, dP)
214
215
216 def testChecksum():
217     # checksum
218     c = randint(0, 2**packet.CHECKSUM_SIZE - 1)
219     eC = packet.Packet.encodeChecksum(c)
220     dC = packet.Packet.decodeChecksum(eC)
221     assert c == dC, (c, eC, dC)
222
223
224 def testInitVector():
225     # init vector
226     iv = randint(0, 2**packet.INIT_VECTOR_SIZE - 1)
227     eIv = packet.Packet.encodeInitVector(iv)
228     dIv = packet.Packet.decodeInitVector(eIv)
229     assert iv == dIv, (iv, eIv, dIv)
230
231
232 def testFrag():
233     # frag
234     fId, fN = (
235         randint(0, 2**packet.FRAGMENT_ID_SIZE - 1),
236         randint(0, 2**packet.FRAGMENT_NUM_SIZE - 1),
237     )
238     eFId, eFN = (
239         packet.Packet.encodeFragmentId(fId),
240         packet.Packet.encodeFragmentNumber(fN),
241     )

```



```

242     dFId, dFN = (
243         packet.Packet.decodeFragmentId(eFId),
244         packet.Packet.decodeFragmentNumber(eFN),
245     )
246     assert (fId, fN) == (dFId, dFN), ((fId, fN), (eFId + eFN), (dFId, dFN))
247
248
249 def testFlags():
250     # flags
251     f = [randint(0, 1) for _ in range(packet.FLAGS_SIZE)]
252     eF = packet.Packet.encodeFlags(f)
253     dF = packet.Packet.decodeFlags(eF)
254     assert f == dF, (f, eF, dF)
255
256
257 def testVersionType():
258     # version type
259     v, pT = (
260         randint(0, 2**packet.VERSION_SIZE - 1),
261         packet.Type(randint(0, max(t.value for t in packet.Type))),
262     )
263     eVt = packet.Packet.encodeVersionType(v, pT)
264     dVt = packet.Packet.decodeVersionType(eVt)
265     assert (v, pT) == dVt, ((v, pT), eVt, dVt)

```

9.3.8 inputimout Original code by Mitsuo Heijo (@johejo). Conatins modification to `inputtimeout.win_inputtimeout` to prevent the automatic appendation of a new line after a timeout.

```

1 from .inputtimeout import inputtimeout, TimeoutOccurred # noqa
2 from __version__ import ( # noqa
3     __version__, __author__, __author_email__, __copyright__, __license__,
4     __description__, __title__, __url__,
5 )

```

```

1 __title__ = 'inputtimeout'
2 __description__ = 'Multi platform standard input with timeout'
3 __url__ = 'http://github.com/johejo/inutimeout'
4 __version__ = '1.0.4'
5 __author__ = 'Mitsuo Heijo'
6 __author_email__ = 'mitsuo_h@outlook.com'

```

```

7 __license__ = 'MIT'
8 __copyright__ = 'Copyright 2018 Mitsuo Heijo'

1 # Modified by @HarryWhitehorn on 2024/04/27:
2 # - Modified win_inputtimeout to prevent automatically appending a newline
3
4 import sys
5
6 DEFAULT_TIMEOUT = 30.0
7 INTERVAL = 0.05
8
9 SP = ' '
10 CR = '\r'
11 LF = '\n'
12 CRLF = CR + LF
13
14
15 class TimeoutOccurred(Exception):
16     pass
17
18
19 def echo(string):
20     sys.stdout.write(string)
21     sys.stdout.flush()
22
23
24 def posix_inputtimeout(prompt='', timeout=DEFAULT_TIMEOUT):
25     echo(prompt)
26     sel = selectors.DefaultSelector()
27     sel.register(sys.stdin, selectors.EVENT_READ)
28     events = sel.select(timeout)
29
30     if events:
31         key, _ = events[0]
32         return key.fileobj.readline().rstrip(LF)
33     else:
34         echo(LF)
35         termios.tcflush(sys.stdin, termios.TCIFLUSH)
36         raise TimeoutOccurred
37
38
39 def win_inputtimeout(prompt='', timeout=DEFAULT_TIMEOUT, newline=False):
40     echo(prompt)

```

```

41     begin = time.monotonic()
42     end = begin + timeout
43     line = ''
44
45     while time.monotonic() < end:
46         if msvcrt.kbhit():
47             c = msvcrt.getwche()
48             if c in (CR, LF):
49                 echo(CRLF)
50                 return line
51             if c == '\003':
52                 raise KeyboardInterrupt
53             if c == '\b':
54                 line = line[:-1]
55                 cover = SP * len(prompt + line + SP)
56                 echo(''.join([CR, cover, CR, prompt, line]))
57             else:
58                 line += c
59         time.sleep(INTERVAL)
60
61     if newline:
62         echo(CRLF)
63     raise TimeoutOccurred
64
65
66 try:
67     import msvcrt
68
69 except ImportError:
70     import selectors
71     import termios
72
73     inputtimeout = posix_inputtimeout
74
75 else:
76     import time
77
78     inputtimeout = win_inputtimeout

```

```

1 MIT License
2
3 Copyright (c) 2017 Mitsuo Heijo
4

```

5 Permission is hereby granted, free of charge, to any person obtaining a copy
6 of this software and associated documentation files (the "Software"), to deal
7 in the Software without restriction, including without limitation the rights
8 to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
9 copies of the Software, and to permit persons to whom the Software is
10 furnished to do so, subject to the following conditions:
11
12 The above copyright notice and this permission notice shall be included in all
13 copies or substantial portions of the Software.
14
15 THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
16 IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
17 FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
18 AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
19 LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
20 OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
21 SOFTWARE.

9.4 Packet Specification

Packets

DEFAULT DEFAULT Packet Specification:

DEFAULT Packet																																		
Offsets	Octet	0								1								2								3								
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	
0	0	Version				Type				Flags								Sequence ID																
4	32	Fragment ID*								Fragment Number*								Init Vector*																
8	64	Checksum Hash*																Data*																

Table 2: *not required

Version

Including a packet version will allow for future changes in specification which out breaking older systems. A recipient must reject a packet if the version does not match the internal version.

Type

The next header is the packet type. This will instruct the recipient on how to unpack the packet. The types are defined as follows:

Types	
Type	Enum
DEFAULT	0
ACK	1
AUTH	2
HEARTBEAT	3
ERROR	4
RESERVED	5..15

For the DEFAULT packet, the packet type is 0 (padded to 4 bits).

Flags

The packet flags are a bit field of the different available flags. Each flag can be toggled independently with some implying the presents of certain headers. The flags are as follows:

Flags								
Octet	1							
Bit	0	1	3	4	2	5	6	7
Flag	RELIABLE	CHECKSUM	COMPRESSED	ENCRYPTED	FRAG	RESERVED		

Sequence ID

The 16-bit sequence ID proved an *unique* identifier for each packet. The Sequence ID is set according to the senders internal value and incremented after each assignment. This allows for the recipient to determine an order to the packets received. The sequence ID must wrap around at 2^{16} back to 0.

Fragment ID

The fragment ID is the position of the fragment in the total packet. This tells the recipient the how to reorder the data of the fragments packets into the final packet. For this header to be present FRAG must be set in the packet's flags.

Fragment Number

The fragment number is the total number of fragments making up the final packet. The recipient will keep collecting fragments until the number of received fragments is equal to the number stated in each fragments fragment number. At this point the final packet can be compiled yielding the full data. For this header to be present FRAG must be set in the packet's flags.

Init Vector

The init vector is 16 bit integer to be used by the recipient when decrypting the packet data. For this header to be present, the ENCRYPTED flag must be set.

Checksum Hash

The checksum is a hash off the packet's data to be checked upon receipt. If the checksum check fails, the recipient should reject the packet. For this header to be present the CHECKSUM flag must be set.

Data

This is the data field of the packet. Upon a valid receipt of a packet, the data is returned to the application layer.

ACK The ACK packet is sent as a reply to any received packets where the RELIABLE flag is set.

ACK packet specification:

ACK Packet																																	
Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
8	64	Default Packet Headers																ACK ID															
12	96	ACK Bits																Data*															

ACK Default Packet Headers

The packet type must be 1 (padded to 4 bits).

ACK ID

The ACK ID is the sequence ID of the packet that the ACK is in acknowledgment of.

ACK Bits

The ACK Bits is a bit field representing the status of the last 16 previous ACKs such that $[ID_{-1}, ID_{-2}, ID_{-3}, \dots, ID_{-17}]$ where ID is equal to the ACK ID of the ACK packet. This help to mitigate against packet loss as each ACK packet also includes an acknowledgment of the last 16 packets (if received).

ACK Data

This is the data field of the packet. This is used during the handshake to send the finished data.

AUTH The AUTH packet is sent during the handshake between a client and server.

AUTH packet specification:

AUTH Packet																																										
Offsets	Octet	0								1								2								3																
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31									
8	64	Default Packet Headers																Reserved								EC Public Key Size (E)																
12	96	EC Public Key																																								
...																																								
10+E	80+E	EC Public Key																Certificate Size (C)*																								
14+E	168+E	EC Public Key																Certificate*																								
...	...	EC Public Key																Certificate*																								
12+C+E	96+C+E	Certificate*																																								

AUTH Default Packet Headers

The packet type must be 2 (padded to 4 bits).

Public Key Size

This is the size *in bytes* of the public key also included in the packet. As different key systems may yield different size keys this tells the recipient where the public key ends when unpacking the packet.

Public Key

This is the public key of the sender which is used in generating the session key.

Certificate Size

This is the size *in bytes* of the certificate also include in the packet. As different certificate systems / encodings may yield different sized certificated this, in conjunction with the public key size, tell the recipient where the certificate ends when unpacking the packet.

Certificate

This is the certificate of the sender. This allows for the recipient to validate the identity of the sender.

HEARTBEAT The HEARTBEAT packet is sent by the server at fixed intervals to check that a client is alive and responding as normal.

HEARTBEAT packet specification:

HEARTBEAT Packet																																	
Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
8	64	Default Packet Headers															Heartbeat								Data*								

HEARTBEAT Default Packet Headers

The packet type must be set to 2 (padded to 4 bits).

Heartbeat

This is a boolean value (padded to 8 bits) dictating the nature of the packet with False and True dictating PING and PONG respectively. If a client receives a PING it must reply with a PONG. A server will expect to get a PONG value back and, after enough failures to reply, will mark a client connection for termination.

HEARTBEAT Data

Unused field.

ERROR The ERROR packet allows for connection members to declare if an error has occurred.

ERROR packet specification:

ERROR Packet																																	
Offsets	Octet	0								1								2								3							
Octet	Bit	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
8	64	Default Packet Headers																Error Major				Error Minor				Data*							

Error Major

This is the Major component of the error being declared.

Error Minor

This is the Minor component of the error being declared. It must be present but can be set to 0 (padded to 4 bits).

ERROR Data

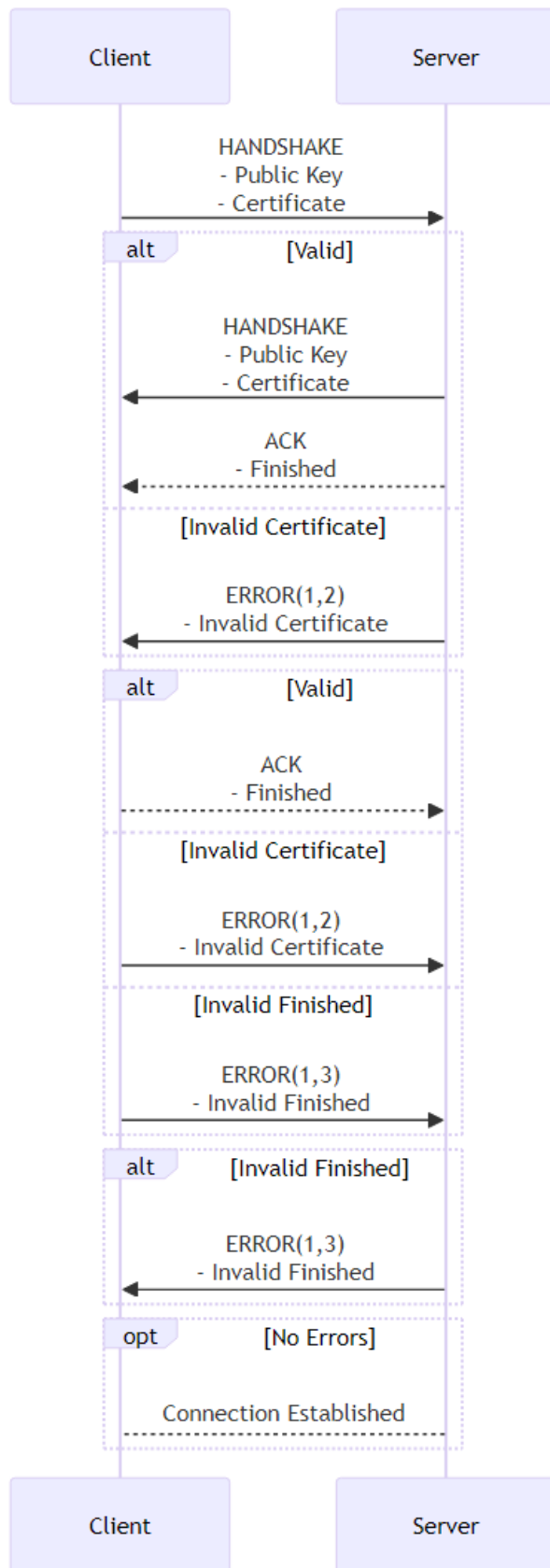
This field allows for additional information about the error to be shared. This used for logging purposes but can also be used for displaying *user-friendly* error messages to a client.

Error Codes The error codes are defined as follows:

Error Codes			
Major	Minor	Name	Description
1	0	CONNECTION	Connection Handshake Could Not Finish
	1	NO SPACE	Server has no more space
	2	CERTIFICATE INVALID	Certificate is invalid / cannot be validated
	3	FINISH INVALID	Finished is invalid
2	0	DISCONNECT	A Party is Disconnecting
	1	SERVER DISCONNECT	The server is closing, all clients must exit gracefully
	2	CLIENT DISCONNECT	The client is closing, the server must handle gracefully
3	0	PACKET	The Packet Cannot be Read
	1	VERSION	The packet version does not match the expected
	2	PACKET TYPE	Unknown / invalid packet type
	3	FLAGS	Unknown / invalid flags
	4	SEQUENCE ID	Sequence id does not match expected
	5	FRAGMENT ID	Unknown / invalid fragment id
	6	FRAGMENT NUMBER	Unknown / invalid fragment number
	7	INIT VECTOR	Unknown / invalid init vector i.e. decrypt fail
	8	COMPRESSION	Decompression fail
	9	CHECKSUM	Unknown / invalid checksum i.e. checksum fail
4..15		RESERVED	

Behavior

Handshake The handshake is always initiated by the client with the client sending an AUTH packet to the server. The server checks the certificate (if present) and either responds with its own AUTH packet or an ERROR packet. The server also sends an ACK packet containing the handshake finished. The client, upon receiving the server's AUTH packet performs the same checks and yields an ERROR packet on a failure. Otherwise, the client also calculates the finished and checks it against the server's version. If it's valid it replies with its own finished ACK otherwise sending an ERROR. The server also checks the client's value for finished and sends an error on failure. If no ERRORS were yielded the parties are considered connected and can begin communication otherwise the connection must be aborted by both parties.



Session Key

Each party generated a session key using their own private key and the other parties public key in a key exchange. This session key is then used in conjunction with the init vector when encrypting and decrypting packet data.

Finished

The finished ACK packet contains a hash generated from the session key. This allows both parties to validate that the other party has the private key for their respective public key as well as acting as a pseudo-checksum to verify the handshake.

Certificate Validation

Certificate validation requires sending a TCP request to the TCP server with the certificate. The TCP server is then able to validate the certificate and yield its success or failure.

Flags Behaviour Reliable

If a packet is sent with the reliable flag, the sender should expect a ACK packet with the relevant ACK ID and should resend the packet until an ACK is received.

When a packet with the reliable flag set is received the recipient should first set its local record of ACKed packet's accordingly and then reply with an ACK packet. If the packet has already been ACKed the data should not be yielded to the application layer.

Checksum

If a packet is sent with the checksum flag set, the sender must calculate a CRC-32 hash with the packet's data and append it to the relevant header.

When a packet with the checksum flag set is received the recipient must recalculate a CRC-32 hash on the packet's data and check that it matches the packets checksum header. On a failure, the recipient must respond with a CHECKSUM ERROR (3.9) and discard the packet.

Compressed

If a packet is sent with the compressed flag set, the sender must perform a compression on the packet's data.

When a packet with the compressed flag set is received the recipient must decompress the packet's data before yielding to the application layer. If the decompression fails, the recipient must respond with a COMPRESSION ERROR (3.8) and discard the packet.

Encrypted

If a packet is sent with the encrypted flag set, the sender must perform an encryption on the packet's data using the session key and a randomly generated init vector. The init vector must then be appended to the relevant packet header.

When a packet with the encrypted flag set is received, the recipient must preform decrypt the packet's data using the session key and the init vector from the packet's headers before yielding the packet's data to the application layer. If the decryption fails, the recipient must respond with a INIT VECTOR ERROR (3.7) and discard the packet.

Frag

If a packet is sent with the frag flag set, the sender must split the data into sub-packets, each with the appropriate fragment id and fragment number.

When a packet with the frag flag set is received, the recipient must buffer the packet. It can then check to see if all other frag packets with the same sequence id have been already buffered by checking the fragment number. If so, the frag packets can be compiled into one packet, using each sub-packet's fragment id and the data can be yielded.

Disconnection Client Disconnect

When a client goes to terminate it must first send a CLIENT DISCONNECT ERROR (2.2) packet. It should then wait for acknowledgement before terminating gracefully. It may terminate after some timeout value if it fails to receive a response.

When a server receives a client disconnect is must acknowledge this before removing the client.

Server Disconnect

When a server goes to terminate it must first send a SERVER DISCONNECT ERROR (2.1) packet. It should then wait for acknowledgement from each client before terminating gracefully. It may terminate after some timeout value if it fails to receive a response from a client.

When a client receives a server disconnect it must acknowledge this before terminating itself (without sending a client disconnect).

9.5 API Specification

Login	Name	Endpoint	Method	Inputs	Returns	Description
Auth						
	Create Account	/auth/register	POST	Username & Password	Account	Register a new Account in the database.
	Get Token	/auth/token	GET		Session Token	Generate and provide a new session token for use by client.
	Get Key	/auth/token	GET		Private Key	Get Account private RSA key.
	Get Cetficate	/auth/certificate	GET		Certificate	Get the server's certificate.
	Validate Certificate	/auth/certificate/validate	GET	Certificate	bool	Validate that a given certificate was singed by the account in the fields.
Games						
	Get Games	/games	GET		list[Game]	Return a list of all available games.
Lobby						
	Get Lobby	/lobby	GET	Lobby ID	Lobby	Return the lobby with the provided lobby ID.
	Get Lobbies	/lobby/all	GET		list[Lobby]	Return all lobbies.
	Create Lobby	/lobby/create	POST	Game Name Game ID	Lobby	Create a new lobby with the given game and return it.
	Find Lobby	/lobby/find	GET	Game Name Game ID	Lobby None	Find a lobby with the given game that has available space
	Get Friends' Lobbies	/lobby/friends	GET		list[Lobby]	Return all lobbies with available space containing an Account registered as a Friend
Friends						
	Get Friends	/friends	GET		list[Friend]	Return a list of all Accounts registered as Friends
	Add Friend	/friends/add	POST	Username	Friend	Register a new Account as a Friend
	Remove Friend	/friends/remove	DELETE	Username		Remove an Account as a Friend

9.6 ERD Diagram

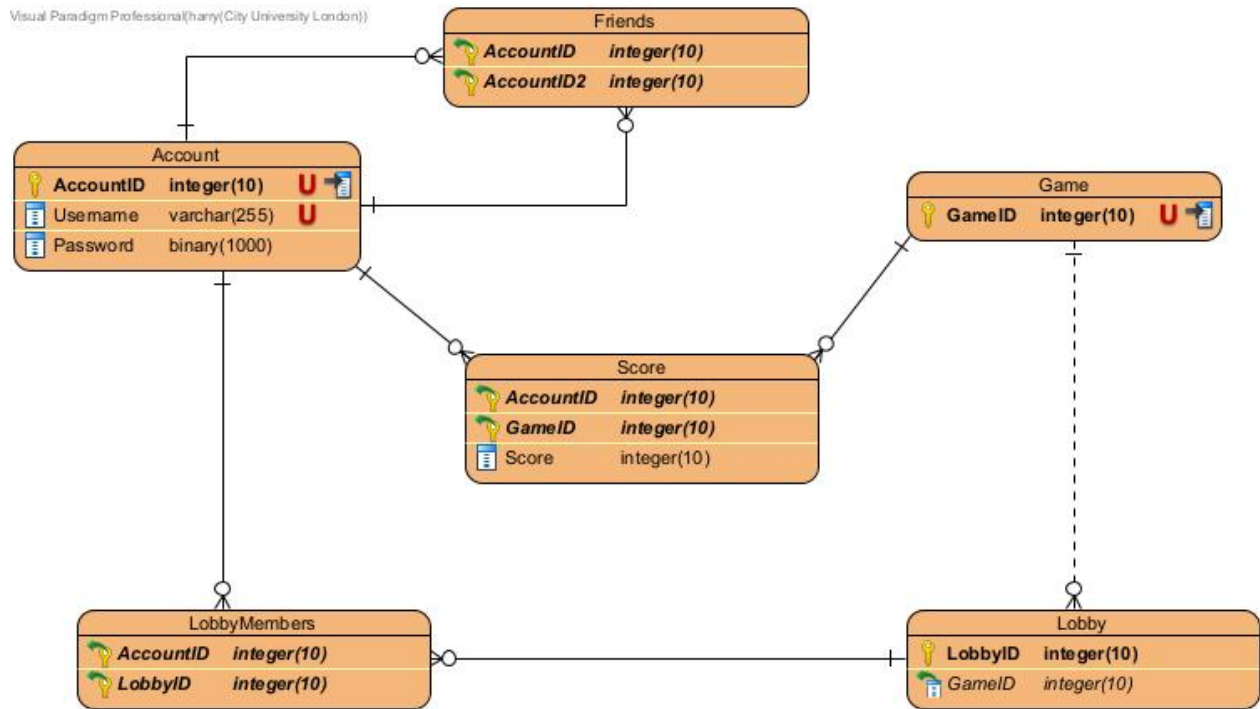


Figure 10: Database Models ERD
