Contents

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, therfore also responsible for authentifcation and certificate validation. Also creates lobbies which include a game server for clients to connect to. Contains the model definitions for use with the database.
Link	
Name	**client**
Type	Python Package
Size	467 lines (after formatting)
Credit	Makes use of a version inputimeout which was modifed to disable the automatic appending of new lines on a timeout.
Description	A cmd line client example for end user use. Handles comunication to RESTful server via dialog with client. Also responsible for creating game client instances. Makes use of inputimeout.
Usage	End user client.
Link	
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 exspected.
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 flexibly, 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:

- Value'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 adherers 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.

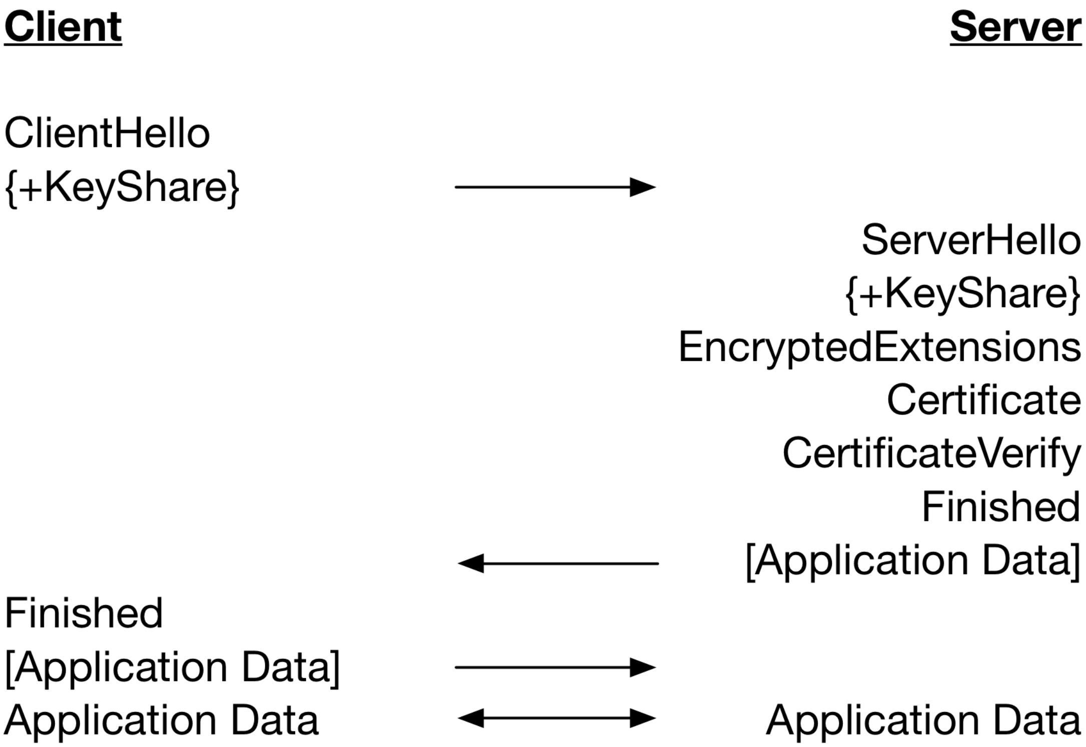


Figure 1: A example of a TLS 1.3 full handshake including a server certificate (wolfSSL, 2019)

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 serval 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 description 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 unpractical 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 defined in the packet specification. As the key chosen for the AUTH packet was EC an Elliptic-curve Diffie-Hellman (ECHD) is used for session key generation.

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

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 `CONST`s 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 `inputimeout` 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.

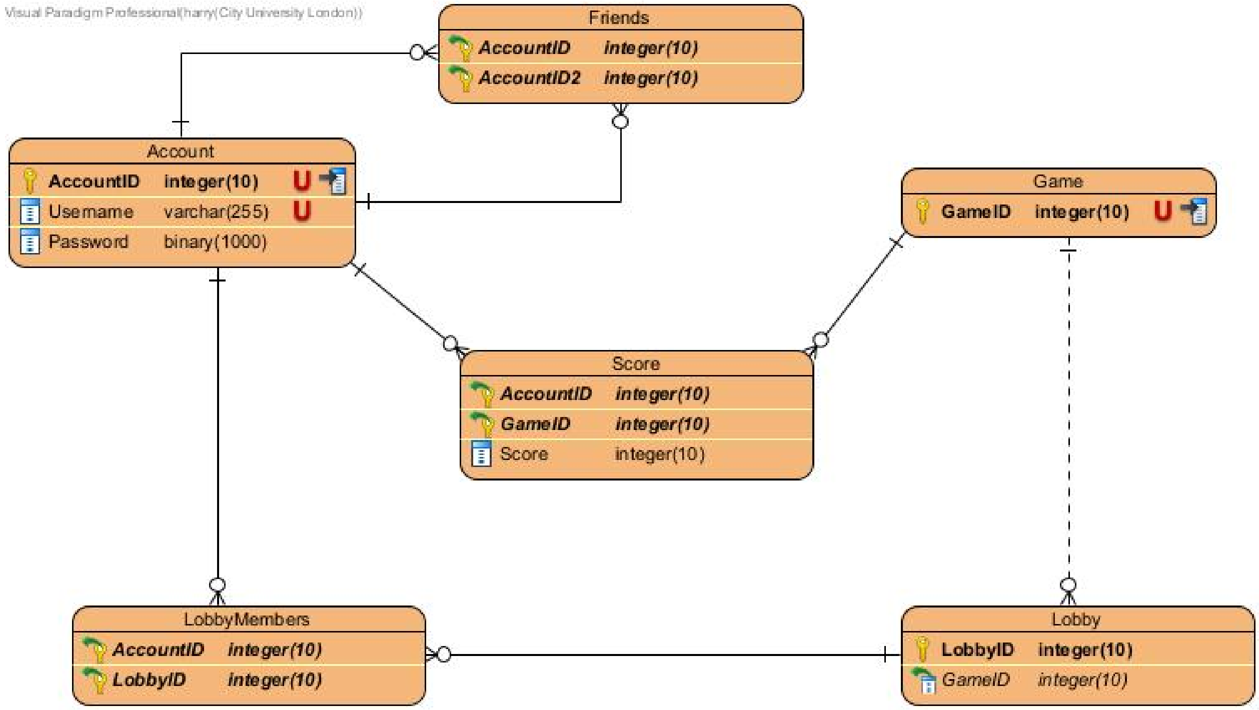


Figure 2: Database Models ERD

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.

```python
from enum import Enum
class Type(Enum):
    DEFAULT = 0
    ACK = 1
    AUTH = 2
    HEARTBEAT = 3
    ERROR = 4

class Flag(Enum):
    RELIABLE = 0
    CHECKSUM = 1
    COMPRESSED = 2
    ENCRYPTED = 3
    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.

```
def sendPacket(self, addr:tuple[str, int], p:packet.Packet) -> None:
        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.

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

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

```python
def sendDefault(self, addr:tuple[str,int]=None, data:bytes|None=None):
    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.

```python
def mainloop(self):
    while True:
        p, addr = self.receivePacket()
        # logic to process and reply (if needed)
        # 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.

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

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

```python
def incrementSequenceId(self) -> None:
    with self.sequenceIdLock:
        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.

```python
def listen(self):
    while self.isRunning.is_set():
        p, addr = self.receivePacket()
        self.receive(p, addr)
```

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

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

The `Server` uses it 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).

```python
def listen(self) -> None:
    while self.isRunning.is_set():
        p, addr = self.receivePacket()
        if p is not None and addr is not None:
            if self.checkClientExists(addr):  # client exists
                if self.getHandshake(addr):  # client handshake complete => allow all packet types
                    self.receive(p, addr)
                else:
                    if p.packet_type in (packet.Type.AUTH, packet.Type.ACK):  # client handshake incomplete => drop all non-AUTH | non-ACK packets
                        self.receive(p, addr)
            else:
                if p.packet_type in (packet.Type.AUTH):  # client not exists => drop all non-AUTH packets
                    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 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.

```python
def sendQueue(self):
        while self.isRunning.is_set():
            addr, p = self.queue.get()
            self.sendPacket(addr, p)
            self.queue.task_done()
            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.

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

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

```python
def queuePacket(self, addr: tuple[str, int], p: packet.Packet) -> None:
    # logic for flags omitted
    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).

```python
def makeClient(self, clientAddr: tuple[str, int]) -> None:
        c = node.Node(
            clientAddr,
            sendLock=self.sendLock,
            socket=self.socket,
        )
        c.outboundThread.start()
        with self.clientsLock:
            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.

```python
def getSequenceId(self, clientAddr: tuple[str, int]) -> int | None:
        with self.clientsLock:
            return (
                self.clients[clientAddr].sequenceId
                if clientAddr in self.clients
                else None
            )
```

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.

```python
def queuePacket(self, addr: tuple[str, int], p: packet.Packet) -> None:
    if p.flags[packet.Flag.RELIABLE.value]:
        self.setSentAckBit(addr, p.sequence_id, False) # set relevant ack bit to False
    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 receival 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 that the recipient has received it.

```python
def sendQueue(self):
    while self.isRunning.is_set():
        addr, p = self.queue.get()
        if p.flags[packet.Flag.RELIABLE.value]:
            if self.getSentAckBit(addr, p): # checks if ACKed
                self.queue.task_done()
                continue # skips
            else:
                self.sendPacket(addr, p) # sends
                self.queue.task_done()
                self.queue.put((addr, p)) # re-adds to the queue
        else:
            self.sendPacket(addr, p)
            self.queue.task_done()
        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*.

```
def handleReliable(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
        if p.flags[packet.Flag.RELIABLE.value]:
            self.setRecvAckBit(addr, p.sequence_id, True) # set relevant recv bit
            self.queueACK(addr, p.sequence_id) # queues and ACK
            return True
        else:
            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.

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

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

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

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

5.4.2.1.4 Receiving an ACK Packet

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

```python
def receiveAck(self, p: packet.AckPacket, addr: tuple[str, int]) -> tuple[packet.Packet, tuple[str, int]]:
    self.setSentAckBit(addr, p.ack_id, True)
    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).

```python
def generateUserCertificate(key) -> x509.Certificate:
    name = [
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, ORG_NAME), # ORG_NAME defined as const e.g. "Paperclip"
        x509.NameAttribute(NameOID.COMMON_NAME, COMMON_NAME), # COMMON_NAME defined as const e.g. "127.0.0.1"
    ]
    subject = issuer = x509.Name(name) # self signed
    cert = (
        x509.CertificateBuilder()
        .subject_name(subject)
        .issuer_name(issuer)
        .public_key(key.public_key())
        .serial_number(x509.random_serial_number())
        .not_valid_before(datetime.datetime.now(datetime.timezone.utc))
        .not_valid_after(
            datetime.datetime.now(datetime.timezone.utc) + datetime.timedelta(days=1) # valid for one day
        )
        .add_extension(
            x509.SubjectAlternativeName([x509.DNSName("localhost")]), # self signed
            critical=False,
        )
        .sign(key, hashes.SHA256())
    )
    return cert
```

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

```
def getDerFromPublicEc(publicKey: ec.EllipticCurvePublicKey) -> bytes:
    ecDer = publicKey.public_bytes(
        encoding=serialization.Encoding.DER,
        format=serialization.PublicFormat.SubjectPublicKeyInfo,
    )
    return ecDer
```

```
def getPublicEcFromDer(publicKeyDer: bytes) -> ec.EllipticCurvePublicKey:
    ec_ = serialization.load_der_public_key(publicKeyDer)
    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.

```
def queueAuth(self, addr: tuple[str, int], cert: Certificate, publicEc: auth.ec.EllipticCurvePublicKey) -> None:
    p = packet.AuthPacket(
        sequence_id=self.getSequenceId(addr), certificate=cert, public_key=publicEc
    )
    self.incrementSequenceId(addr)
    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.

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

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.

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

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

```python
def generateFinished(sessionKey: bytes, finishedLabel: bytes, messages: bytes):
    hashValue = hashes.Hash(hashes.SHA256())
    hashValue.update(messages)
    hashValue = hashValue.finalize()
    prf = hmac.HMAC(sessionKey, hashes.SHA256())
    prf.update(finishedLabel)
    prf.update(hashValue)
    prf = prf.finalize()
    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.

```python
def connect(self) -> None:
    self.outboundThread.start() # start outbound
    self.queueAuth(self.targetAddr, self.cert, self.ecKey.public_key()) # send auth
    authPacket = None
    ackPacket = None
    while True:
        p, addr = self.receivePacket()
        if p is not None:
            # logic
            if p.packet_type == packet.Type.AUTH: # AUTH packet -> generate session key, validate certificate, queueFinished
                authPacket = p
                self.sessionKey = auth.generateSessionKey(
                    self.ecKey, p.public_key
                )
                if not self.validateCertificate(p.certificate):
                    # certificate not valid
                    # abort
                    break
                self.queueFinished(
                    self.targetAddr, p.sequence_id, self.sessionKey
                )
            elif p.packet_type == packet.Type.ACK: # ACK packet
                ackPacket = p
                self.receiveAck(p, addr)
            if authPacket is not None and ackPacket is not None: # wait until both parts received
                break
        else:
            # Server not responsive
            # abort
            break
    if self.validateHandshake(ackPacket.data): # check finished
        # success
        self.inboundThread.start() # start inbound
    else:
        # 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`).

```python
def receiveAuth(self, p: packet.AuthPacket, addr: tuple[str, int]) -> tuple[packet.AuthPacket, tuple[str, int]]:
    if addr not in self.clients:  # new client
        if self.isNotFull():  # check space
            valid, accountId = self.validateCertificate(p.certificate)
            if not valid:
                # invalid certificate
                # abort
                return
            else:
                self.makeClient(addr, p.certificate, accountId)
                self.regenerateEcKey(addr)
                sessionKey = auth.generateSessionKey(
                    self.getEcKey(addr), p.public_key
                )
                self.setSessionKey(addr, sessionKey) # sets client sessionKey for later reference
                self.queueAuth(addr, self.cert, self.getEcKey(addr).public_key())
                self.queueFinished(addr, p.sequence_id, self.getSessionKey(addr))
        else:
            # no space
            # abort
            return
    else:
        sessionKey = auth.generateSessionKey(self.getEcKey(addr), p.public_key)
    if addr in self.clients: # existing client
        if self.getSessionKey(addr) != sessionKey:  # new client sessionKey
            valid, accountId = self.validateCertificate(p.certificate)
            if not valid:
                # invalid certificate
                # abort
                # remove client
                return
            else:
                self.regenerateEcKey(addr)
                sessionKey = auth.generateSessionKey(
                    self.getEcKey(addr), p.public_key
                )
                self.setSessionKey(addr, sessionKey)  # make new session key
                self.queueAuth(addr, self.cert, self.getEcKey(addr).public_key())
                self.queueFinished(addr, p.sequence_id,
                    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.

```
def receiveAck(self, p: packet.AckPacket, addr: tuple[str, int]) -> None:
        super().receiveAck(p, addr)
        if p.data is not None and not self.getHandshake(addr):  # ack has payload & client has not completed handshake => validate handshake
            if not self.validateHandshake(addr, p.data): # checks and sets the clients handshake
                # invalid finish
                # abort
                return
            else:
                # success
                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.

```
def generateCipher(sessionKey: bytes, iv: bytes = generateInitVector()) -> tuple[Cipher, bytes]:
    cipher = Cipher(algorithms.AES(sessionKey), modes.CBC(iv))
    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.

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

```python
    )
    cipher, iv = auth.generateCipher(session_key, iv)
    self.init_vector = iv # assign to header
    self.data = auth.encryptBytes(cipher, self.data)
```

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

```python
def encryptBytes(cipher: Cipher, rawBytes: bytes, autoPad=True) -> bytes:
    if autoPad:
        padder = padding.PKCS7(algorithms.AES.block_size).padder()
        rawBytes = padder.update(rawBytes) + padder.finalize()
    encryptor = cipher.encryptor()
    encryptedBytes = encryptor.update(rawBytes) + encryptor.finalize()
    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.

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

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

```python
def decryptBytes(cipher: Cipher, encryptedBytes: bytes, autoUnpad: bool = True) -> bytes:
    decryptor = cipher.decryptor()
    decryptedBytes = decryptor.update(encryptedBytes) + decryptor.finalize()
    if autoUnpad:
        unpadder = padding.PKCS7(algorithms.AES.block_size).unpadder()
        decryptedBytes = unpadder.update(decryptedBytes) + unpadder.finalize()
    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.

```
def compressData(self) -> None:
    self.flags[Flag.COMPRESSED.value] = 1 # ensure flag set
    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 enure that no header or checksum is appended to the bytes as this would create unnecessary overhead.

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

5.4.2.4.2.2 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.

```
def decompressData(self) -> None:
    if self.flags[Flag.COMPRESSED.value]:
        self.data = utils.decompressData(self.data)
    else:
        # 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.

```
def decompressData(data: bytes) -> bytes:
    # no header or checksum
    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.

```
def generateChecksum(data: bytes) -> int:
    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.

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

```python
    data = self.data if self.data is not None else b"" # sets to empty byte string if None
    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.

```python
def validateChecksum(self) -> bool:
    if self.flags[Flag.CHECKSUM.value]:
        data = self.data if self.data is not None else b"" # sets to empty byte string if None
        return self.checksum == utils.generateChecksum(data)
    else:
        # 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 serval sub-packages. These are then resembled 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 packets data into fragmented chunks and creates a list of *fragment* packets.

```python
def fragment(self):
    self.flags[Flag.FRAG.value] = 1 # ensure flag set
    header = Packet._getHeader(self) # returns dictionary of packet's headers (where set)
    fragData = utils.fragmentData(self.data)
    fragment_number = len(fragData)
    return [
        self._createFragment(
            header, fragment_id=i, fragment_number=fragment_number, data=data # set fragment_id, fragment_number and data through comprehension
        )
        for i, data in enumerate(fragData)
    ]
```

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

```python
@classmethod
def _createFragment(
    cls, header: dict, fragment_id: int, fragment_number: int, data: bytes
):
    return cls(
        **header,
        fragment_id=fragment_id,
        fragment_number=fragment_number,
        data=data,
    )
```

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.

```python
def fragmentData(data: bytes) -> list[bytes]:
    return [
        data[i : i + MAX_FRAGMENT_SIZE] for i in range(0, len(data), MAX_FRAGMENT_SIZE)
    ]
```

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.

```python
def handleFrag(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
    if p.flags[packet.Flag.FRAG.value]:
        if p.sequence_id not in self.getFragBuffer(addr): # new fragment sequence id
            self.getFragBuffer(addr)[p.sequence_id] = [
                None for _ in range(p.fragment_number) # Empty list with size == p.fragment_number
            ]
        self.getFragBuffer(addr)[p.sequence_id][p.fragment_id] = p
        if all(self.getFragBuffer(addr)[p.sequence_id]): # all list members not None
            defrag = p.defragment(self.getFragBuffer(addr)[p.sequence_id])
            del self.getFragBuffer(addr)[p.sequence_id] # remove fragment sequence id from dict
            self.receive(defrag, addr)
        return True
    else:
        return False
```

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

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

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

```python
def defragmentData(fragments: list[bytes]) -> bytes:
    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`.

```python
def queuePacket(self, addr: tuple[str, int], p: packet.Packet) -> None:
    # reliable -> checksum -> compress -> encrypt -> frag
    if p.flags[packet.Flag.RELIABLE.value]:
        self.setSentAckBit(addr, p.sequence_id, False)
    if p.flags[packet.Flag.CHECKSUM.value]:
        p.setChecksum()
    if p.flags[packet.Flag.COMPRESSED.value]:
        p.compressData()
    if p.flags[packet.Flag.ENCRYPTED.value]:
        p.encryptData(self.getSessionKey(addr))
    if p.flags[packet.Flag.FRAG.value]:
        frags = p.fragment()
        for frag in frags:
            self.getQueue(addr).put((addr, frag)) # queue each fragment
    else:
        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.

```python
def handleFlags(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
    # defrag -> decrypt -> decompress -> validate checksum -> reliable
    if self.handleFrag(p, addr):
        return False
    else:
        self.handleEncrypted(p, addr)
        self.handleCompressed(p, addr)
        self.handleChecksum(p, addr)
        self.handleReliable(p, addr)
        return True
```

5.4.3 Iteration 3

5.4.3.1 ACK Bits and Rolling Reset The Node class utilizes it 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 revives 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.

```python
def receiveAck(self, p: packet.AckPacket, addr: tuple[str, int]) ->
   tuple[packet.Packet, tuple[str, int]]:
    self.setNewestSeqId(
        addr, self.getNewerSeqId(self.getNewestSeqId(addr), p.sequence_id)
    )
    self.setSentAckBit(addr, p.ack_id, True)
    # set all bits from ack bits to true (to mitigate lost ack)
    for i, j in enumerate(range(p.ack_id - 1, p.ack_id - 1 -
       packet.ACK_BITS_SIZE, -1)):
        if p.ack_bits[i]:
            self.setSentAckBit(addr, j, True)
    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.

```python
def getNewerSeqId(currentSeqId: int, newSeqId: int) -> int:
    currentDiff = (newSeqId - currentSeqId) % (2**16)
    newDiff = (currentSeqId - newSeqId) % (2**16)
    if newDiff < currentDiff:
        return currentSeqId
    else:
        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`.

```
def handleReliable(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
    if p.flags[packet.Flag.RELIABLE.value]:
        self.setNewestSeqId(
            addr, self.getNewerSeqId(self.getNewestSeqId(addr), p.sequence_id)
        )
        self.setRecvAckBit(addr, p.sequence_id, True)
        self.resetRecvAckBits(addr)
        self.queueACK(addr, p.sequence_id)
        return True
    else:
        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.

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

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

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

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

```python
            sequence_id=self.getSequenceId(addr),
            flags=flags,
            heartbeat=heartbeat,
            data=data,
        )
        self.incrementSequenceId(addr)
        self.queuePacket(addr, p)
```

The `heartbeatThread` uses the `heartbeat` method.

```python
def startThreads(self) -> None:
        super().startThreads()
        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.

```python
def heartbeat(self) -> None:
        while self.isRunning.is_set():
            time.sleep(HEARTBEAT_MIN_TIME)
            with self.clientsLock:
                clients = [k for k in self.clients.keys()]
            for clientAddr in clients:
                heartbeat = self.getHeartbeat(clientAddr)
                delta = (datetime.now() - heartbeat).seconds
                if delta > HEARTBEAT_MAX_TIME:
                    self.removeClient(
                        clientAddr,
                        debugStr=f"due to heartbeat timeout (last contact was {heartbeat})",
                    )
                elif delta > HEARTBEAT_MIN_TIME:
                    self.queueHeartbeat(clientAddr, heartbeat=False)
```

```python
def removeClient(self, clientAddr: tuple[str, int], debugStr="") -> None:
    if self.checkClientExists(clientAddr):
        cId = self.getClientId(clientAddr)
        with self.clientsLock:
            self.clients[clientAddr].isRunning.clear()
            del self.clients[clientAddr]
            if self.onClientLeave:
                self.onClientLeave(clientAddr, cId)
```

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

```python
def receiveHeartbeat(
        self, p: packet.HeartbeatPacket, addr: tuple[str, int]
    ) -> tuple[packet.Packet, tuple[str, int]]:
        if not p.heartbeat:
            self.queueHeartbeat(addr, heartbeat=True)
            pass
        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.

```python
class Major(Enum):
    ERROR = 0
    CONNECTION = 1
    DISCONNECT = 2
    PACKET = 3

class Minor(Enum): pass

class PaperClipError(Exception): """Unknown error"""
```

The three Major error types then inherent 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.

```python
# connection
class ConnectionErrorCodes(Minor):
    CONNECTION = 0
    NO_SPACE = 1
    CERTIFICATE_INVALID = 2
    FINISH_INVALID = 3

class ConnectionError(PaperClipError): """Handshake connection could not be finished"""

class NoSpaceError(ConnectionError): """Server has insufficient space to accept new clients"""

class CertificateInvalidError(ConnectionError): """Certificate is invalid / can not be validated"""

class FinishInvalidError(ConnectionError): """Finish is invalid"""

_connectionErrors = {
    ConnectionErrorCodes.CONNECTION: ConnectionError,
    ConnectionErrorCodes.NO_SPACE: NoSpaceError,
    ConnectionErrorCodes.CERTIFICATE_INVALID: CertificateInvalidError,
    ConnectionErrorCodes.FINISH_INVALID: FinishInvalidError,
}

def getConnectionError(minor: ConnectionErrorCodes | int) -> ConnectionError:
    try:
        minor = minor if isinstance(minor, Minor) else ConnectionErrorCodes(minor)
        if minor in _connectionErrors:
            return _connectionErrors[minor]
        else:
            return PaperClipError
    except ValueError:
        return PaperClipError

def getConnectionCode(error: ConnectionError) -> ConnectionErrorCodes:
    try:
        return list(_connectionErrors.keys())[
            list(_connectionErrors.values()).index(error)
        ]
    except ValueError:
        return PaperClipError
```

Convenience methods allow for conversion between `Enums` and `PaperClipErrors`. The getError method takes, either Enum or integer, Major and Minor codes are returns the relevant Exception.

```python
def getError(major: Major | int, minor: Minor | int = 0) -> PaperClipError:
    try:
        major = major if isinstance(major, Major) else Major(major)
        match major:
            case Major.CONNECTION:
                return getConnectionError(minor)
            case Major.DISCONNECT:
                return getDisconnectError(minor)
            case Major.PACKET:
                return getPacketError(minor)
            case _:
                return PaperClipError
    except TypeError:
        return PaperClipError
```

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

```python
def getMinor(major: Major, minor: int) -> Minor:
    match major:
        case Major.CONNECTION:
            return ConnectionErrorCodes(minor)
        case Major.DISCONNECT:
            return DisconnectErrorCodes(minor)
        case Major.PACKET:
            return PacketErrorCodes(minor)
        case _:
            return Minor
```

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

```python
def getErrorCode(error: PaperClipError) -> tuple[Major, Minor]:
    match error:
        case c if issubclass(c, ConnectionError):
            return (Major.CONNECTION, getConnectionCode(error))
        case d if issubclass(d, DisconnectError):
            return (Major.DISCONNECT, getDisconnectCode(error))
        case p if issubclass(p, PacketError):
            return (Major.PACKET, getPacketCode(error))
        case _:
            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.

```python
def queueError(
    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
    ) -> None:
    sId = self.getSequenceId(addr)
    p = packet.ErrorPacket(
        sequence_id=sId if sId is not None else 0,
        flags=flags,
        major=major,
        minor=minor,
        data=data,
    )
    if sId is not None:
        self.incrementSequenceId(addr)
    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.

```python
def receivePacket(self,) -> tuple[packet.Packet, tuple[str, int]] | tuple[None, None]:
    data, addr = self.socket.recvfrom(SOCKET_BUFFER_SIZE)
    try:
        p = packet.unpack(data) # unpacking can yield a PacketError
        return p, addr
    except error.PacketError as e:
        major, minor = error.getErrorCod(e)
        self.queueError(addr, major, minor)
        return None, None
```

```python
def receive(self, p: packet.Packet, addr: tuple[str, int]) -> tuple[packet.Packet, tuple[str, int]] | None:
    if p is not None:
        if self.handleFlags(p, addr):
            match p.packet_type:
                # packet type cases omitted for clarity
                case _: # unknown packet type
                    self.queueError(
                        addr,
                        major=error.Major.PACKET,
                        minor=error.PacketErrorCodes.PACKET_TYPE,
                        data=p.sequence_id,
                    )
```

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.

```
case packet.Type.ERROR:
    try:
        return self.receiveError(p, addr)
    except error.PaperClipError as e:
        self.handleError(p, addr, e)
```

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

```
def handleError(self, p: packet.ErrorPacket, addr: tuple[str, int], e: error.PaperClipError) -> None:
    match e:
        case error.ConnectionError():
            self.handleConnectionError(p, addr, e)
        case error.DisconnectError():
            self.handleDisconnectError(p, addr, e)
        case error.PacketError():
            self.handlePacketError(p, addr, e)
        case _:
            raise e
```

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

```
    def handleConnectionError(self, p: packet.ErrorPacket, addr: tuple[str, int], e: error.ConnectionError) -> None:
        match e:
            case error.NoSpaceError():
                return self.quit("no server space", e)
            case error.CertificateInvalidError():
                return self.quit("invalid certificate", e)
            case error.FinishInvalidError():
                return self.quit("invalid finish", e)
            case _:
                raise e
```

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

```python
    def handleDisconnectError(
        self, p: packet.ErrorPacket, addr: tuple[str, int], e: error.DisconnectError) -> None:
        match e:
            case error.ServerDisconnectError:
                pass  # overwrite
            case error.ClientDisconnectError:
                pass  # overwrite
            case _:
                raise e
```

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

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

```python
def queueDisconnect(self, addr: tuple[str, int], flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)], data: bytes | None = None) -> None:
    self.queueError(
        addr,
        flags=flags,
        major=error.Major.DISCONNECT,
        minor=error.DisconnectErrorCodes.DISCONNECT,
        data=data,
    )
```

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.

```python
def handleDisconnectError(
        self, p: packet.ErrorPacket, addr: tuple[str, int], e: error.DisconnectError
    ) -> None:
        match e:
            case error.ServerDisconnectError():
                self._quit(e)
            case error.ClientDisconnectError():
                pass  # should not react to client disconnect
            case _:
                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.

```python
def handleDisconnectError(
        self, p: packet.ErrorPacket, addr: tuple[str, int], e: error.DisconnectError
    ) -> None:
        match e:
            case error.ServerDisconnectError():
                pass  # should not react to server disconnect
            case error.ClientDisconnectError():
                self.removeClient(addr, "The client has closed")
            case _:
                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).

```python
def queueDisconnect(self, flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)], data: bytes | None = None):
    with self.clientsLock:
        clientAddrs = [addr for addr in self.clients]
    for addr in clientAddrs:
        self.queueError(
            addr,
            flags=flags,
            major=error.Major.DISCONNECT,
            minor=error.DisconnectErrorCodes.SERVER_DISCONNECT,
            data=data,
        )
```

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.

```
# .env

# udp
S_HOST=127.0.0.1
S_PORT=2024
C_HOST=127.0.0.1
C_PORT=2025
## node
SOCKET_BUFFER_SIZE = 1024
SEND_SLEEP_TIME = 0.1
QUEUE_TIMEOUT = 10
SOCKET_TIMEOUT = 20
## server
HEARTBEAT_MAX_TIME = 120
HEARTBEAT_MIN_TIME = 30
MAX_CLIENTS
## auth
ORG_NAME = Paperclip
COMMON_NAME = 127.0.0.1
## utils
MAX_FRAGMENT_SIZE = 988

# client
TCP_PORT = 5000

# app
FLASK_APP = server
PRUNE_TIME = 58
SECRET_KEY = MyVerySecretKey
SQLALCHEMY_DATABASE_URI = mysql://root:root@localhost:3306/paperclip

# debug
DEBUG = True
```

The variables can then be loaded from the `os.environ` by first calling `dotenv.load_dot(".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)

```python
import os
import dotenv

dotenv.load_dotenv(".env")
S_HOST = os.environ.get("S_HOST")
```

```
S_PORT = int(os.environ.get("S_PORT"))
C_HOST = os.environ.get("C_HOST")
C_PORT = int(os.environ.get("C_PORT"))
# node
SOCKET_BUFFER_SIZE = int(os.environ.get("SOCKET_BUFFER_SIZE"))
SEND_SLEEP_TIME = float(os.environ.get("SEND_SLEEP_TIME"))
QUEUE_TIMEOUT = int(os.environ.get("QUEUE_TIMEOUT"))
SOCKET_TIMEOUT = int(os.environ.get("SOCKET_TIMEOUT"))
# server
HEARTBEAT_MAX_TIME = int(os.environ.get("HEARTBEAT_MAX_TIME"))
HEARTBEAT_MIN_TIME = int(os.environ.get("HEARTBEAT_MIN_TIME"))
MAX_CLIENTS = (
    int(os.environ.get("MAX_CLIENTS"))
    if os.environ.get("MAX_CLIENTS") is not None
    else float("inf")
)
# auth
ORG_NAME = os.environ.get("ORG_NAME")
COMMON_NAME = os.environ.get("COMMON_NAME")
# utils
MAX_FRAGMENT_SIZE = int(os.environ.get("MAX_FRAGMENT_SIZE"))
```

5.4.4.2 Logging A `logging.Logger` is used to provided 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.

```
import logging
import sys

class bcolors:
    HEADER = "\033[95m"
    OKBLUE = "\033[94m"
    OKCYAN = "\033[96m"
    OKGREEN = "\033[92m"
    WARNING = "\033[93m"
    FAIL = "\033[91m"
    ENDC = "\033[0m"
    BOLD = "\033[1m"
    UNDERLINE = "\033[4m"

logger = logging.getLogger(__name__)
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.

```python
printHandler = logging.StreamHandler(sys.stdout)
printHandler.setLevel(logging.INFO)
printHandler.setFormatter(
    logging.Formatter(f"{bcolors.OKBLUE}%(threadName)s{bcolors.ENDC} - %(message)s")
)
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.

```python
class ColorFilter(logging.Filter):
    colorCodes = [
        getattr(bcolors, attr) for attr in dir(bcolors) if not attr.startswith("__")
    ]

    def filter(self, record: logging.LogRecord) -> bool:
        for color in self.colorCodes:
            record.msg = record.msg.replace(color, "")
        return True

fileHandler = logging.FileHandler("paperclip.log")
fileHandler.setLevel(logging.DEBUG)
fileHandler.addFilter(ColorFilter())
fileHandler.setFormatter(
    logging.Formatter("%(asctime)s - %(levelname)s - %(threadName)s - %(message)s")
)
logger.addHandler(fileHandler)
```

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

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

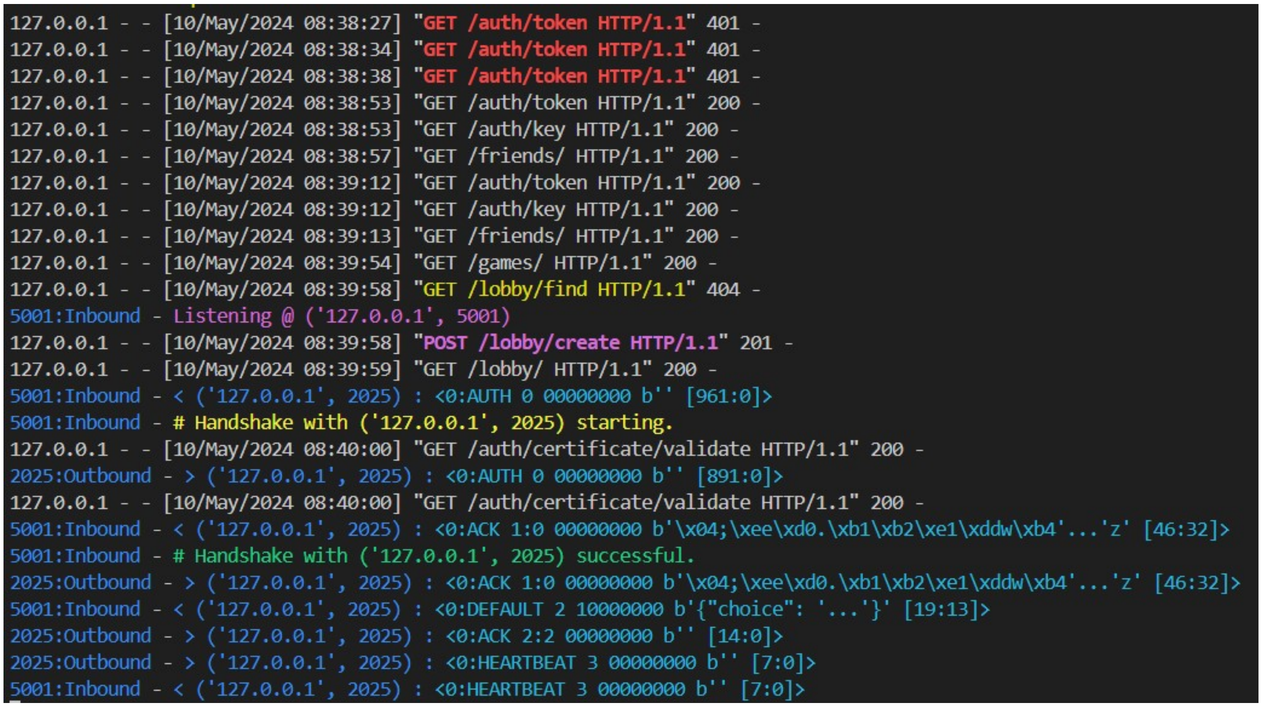
```python
logger.warning(f"\tInvalid checksum: {p}") # WARNING: log invalid checksum
```

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

```python
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.

```python
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`.

```python
uri = os.environ.get("SQLALCHEMY_DATABASE_URI") # get uri from .env
_init = False
if not database_exists(uri): # create database if not exists
    _init = True
    create_database(uri)
app.config["SQLALCHEMY_DATABASE_URI"] = uri

db.init_app(app)

with app.app_context():
    db.create_all() # create all tables

if _init: # if database was created
    with app.app_context(): # create some dummy data
        # init games
        from rps import ID, NAME, MIN_PLAYERS, MAX_PLAYERS
        Statement.createGame(ID, NAME, MIN_PLAYERS, MAX_PLAYERS)
        # example accounts
        m = Statement.createAccount("Mario", "ItsAMe123")
        p = Statement.createAccount("Peach", "MammaMia!")
        b = Statement.createAccount("Bowser", "M4r10SucK5")
        Statement.createFriends(m.id, p.id)
        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.

```
class Friends(db.Model):
    account_one_id = db.Column(
        db.Integer, db.ForeignKey("account.id"), primary_key=True
    )
    account_two_id = db.Column(
        db.Integer, db.ForeignKey("account.id"), primary_key=True
    )
```

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

```
class Statement:
    @staticmethod
    def getFriends(accountId: int) -> list[Account]: # retrieve list of Accounts who are Friends with id
        friends = Friends.query.filter( # filter where either account_one_id or account_two_id is accountId
            (Friends.account_one_id == accountId)
            | (Friends.account_two_id == accountId)
        )
        friends = [ # get the account_id of the other account
            friend.account_one_id
            if friend.account_one_id != accountId
            else friend.account_two_id
            for friend in friends
        ]
        friends = [Statement.getAccount(id) for id in friends] # get list of accounts
        return friends

    @staticmethod
    def createFriends(accountIdOne: int, accountIdTwo: int) -> Friends: # create, commit and return Friends
        # enure that idOne < idTwo for index efficiency & easier look-up
        idOne = min(accountIdOne, accountIdTwo)
        idTwo = max(accountIdOne, accountIdTwo)
        friends = Friends(account_one_id=idOne, account_two_id=idTwo)
        db.session.add(friends)
        db.session.commit()
        return friends

    @staticmethod
    def removeFriends(accountIdOne: int, accountIdTwo: int) -> bool: # delete Friends. True on success.
        # ensure that idOne < idTwo
        idOne = min(accountIdOne, accountIdTwo)
        idTwo = max(accountIdOne, accountIdTwo)
        friends = Friends.query.filter(
            (Friends.account_one_id == idOne) & (Friends.account_two_id == idTwo)
        )
        if friends is not None:
            friends.delete() # delete
            db.session.commit()
            return True
        else:
            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.

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

```python
class Statement:
    @staticmethod
    def getGame(gameId: int) -> Game:
        return Game.query.filter_by(id=gameId).scalar() # retrieve Game by id

    @staticmethod
    def getGames() -> list[Game]:
        return Game.query.all() # retrieve all Game

    @staticmethod
    def createGame(id:int, name:str, min_players:int, max_players:int) -> Game: # create, commit and return Game
        game = Game(id=id, name=name, min_players=min_players, max_players=max_players)
        db.session.add(game)
        db.session.commit()
        return game

    @staticmethod
    def findGame(gameName: str) -> Game | None:
        return Game.query.filter_by(name=gameName).scalar() # retrieve Game 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.

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

    def hashPassword(self, password: str) -> None:
        self.password = generate_password_hash(password)

    def verifyPassword(self, password: str) -> bool:
        return check_password_hash(self.password, password)

    def generateKey(self, password: bytes) -> None:
        k = auth.generateRsaKey()
        self.private_key = auth.getDerFromRsaPrivate(k, password) # encrypts DER with password for security
        self.public_key = auth.getDerFromRsaPublic(k.public_key())

    @staticmethod
    def decryptKey(self, key: bytes, password: bytes) -> auth.rsa.RSAPublicKey:
        k = auth.getRsaPrivateFromDer(key, password)
        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.

```
class Statement:
    # get
    @staticmethod
    def getAccount(userId: int) -> Account:
        return Account.query.filter_by(id=userId).scalar() # retrieve Account by id

    # create
    @staticmethod
    def createAccount(username: str, password: str) -> Account: # create, commit and return Account
        account = Account(username=username)
        account.hashPassword(password) # hash password
        account.generateKey(password.encode()) # generate RSA key
        db.session.add(account)
        db.session.commit()
        return account

    # find
    @staticmethod
    def findAccount(username: str) -> Account | None:
        return Account.query.filter_by(username=username).scalar() # retrieve 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.

```python
def isNotFull(self) -> bool:
    return self.gameServer.isNotFull()

def isEmpty(self) -> bool:
    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.

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

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

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

def _heartbeatDelta(self) -> int:
        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().

```python
def onJoin(self, addr: tuple[str, int], accountId: int) -> None:
    self.members.append(accountId)
    self.heartbeat = True

def onLeave(self, addr: tuple[str, int], accountId: int) -> None:
    self.members.remove(accountId)
    if self.isEmpty():
        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.

```python
auth = HTTPBasicAuth()

@auth.verify_password
def verifyPassword(username: str, password: str) -> bool:
    account = Statement.validateToken(username)  # check token
    if not account:  # if token not valid
        account = Statement.findAccount(username=username)  # check account
        if not account or not account.verifyPassword(
            password
        ):  # if account not exist or wrong password
            return False
    g.account = account # store (until overwrite) in flask globals
    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).

```python
def generateToken(self, expiration: int = 600) -> str:
        data = {
            "id": self.id,
            "exp": datetime.datetime.now() +
               datetime.timedelta(seconds=expiration),
        }
        token = jwt.encode(data, current_app.config["SECRET_KEY"],
           algorithm="HS256")
        return token
```

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

```python
class Statement:
    @staticmethod
    def validateToken(token: str) -> Account | None:
        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.

```python
@staticmethod
def validateToken(token: str):
    try:
        data = jwt.decode(
            token,
            current_app.config["SECRET_KEY"],
            leeway=datetime.timedelta(seconds=10),
            algorithms=["HS256"],
        )
    except:
        return None
    account = Statement.getAccount(data.get("id"))
    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.

```python
@main.route("/auth/register", methods=["POST"])
def createAccount():
    username = request.json.get("username")
    password = request.json.get("password")
    if not (username or password):  # check not null
        abort(400)  # missing args
    if Statement.findAccount(username):  # check if account exists
        abort(400)  # account already exists
    account = Statement.createAccount(username, password)
    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`.

```python
@main.route("/auth/token")
@auth.login_required
def getAuthToken():
    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`.

```python
@main.route("/auth/key")
@auth.login_required
def getKey():
    return jsonify(
        {
            "key": base64.encodebytes(g.account.private_key).decode(),
            "account-id": g.account.id,
        }
    )
```

5.4.4.4.2.2 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.

```python
@main.route("/friends/")
@auth.login_required
def getFriends():
    friends = Statement.getFriends(g.account.id)
    return jsonify(
        {
            "friends": [
                {"id": account.id, "username": account.username} for account in friends
            ]
        }
    )
```

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.

```python
@main.route("/friends/add", methods=["POST"])
@auth.login_required
def addFriend():
    username = request.json.get("username")
    if username is None:
        abort(400)  # missing args
    account = g.account
    other = Statement.findAccount(username)
    if other is None:
        abort(404)
    Statement.createFriends(account.id, other.id)
    return jsonify(
        {
            "account": {"id": account.id, "username": account.username},
            "other": {"id": other.id, "username": other.username},
        }
    ), 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

```python
@main.route("/friend/remove", methods=["DELETE"])
@auth.login_required
def removeFriend():
    username = request.json.get("username")
    if username is None:
        abort(400)  # missing args
    account = g.account
    other = Statement.findAccount(username)
    if other is None:
        abort(404) # no such account
    success = Statement.removeFriends(account.id, other.id)
    if success:
        return jsonify(data=[]), 204
    else:
        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.

```
@main.route("/games/")
@auth.login_required
def getGames():
    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 acceptes **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`.

```
@main.route("/lobby/")
@auth.login_required
def getLobby():
    lobbyId = request.json.get("lobby-id")
    if not lobbyId:
        abort(400)  # missing args
    lobby = lobbyHandler.getLobby(lobbyId)
    return jsonify(
        {"lobby-id": lobby.id, "lobby-addr": lobby.getAddr(), "game-id":
           lobby.gameId}
    )
```

The `getLobbies` method at `/lobby/all` complies all lobbies currently in the `LobbyHandler`. It the 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`.

```
@main.route("/lobby/all")
@auth.login_required
def getLobbies():
    lobbies = LobbyHandler.getAll()
    games = {game.id: game.name for game in Statement.getGames()}
    data = lambda lobby: {
        "game": {"game-id": lobby.game_id, "game-name": games[lobby.game_id]},
        "size": Statement.getLobbySize(lobby.id),
        "is-full": Statement.getIsLobbyFree(lobby.id),
    }
    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.

```python
@main.route("/lobby/find")
@auth.login_required
def findLobby():
    gameId = request.json.get("game-id")
    gameName = request.json.get("game-name")
    if not (gameId or gameName):  # check args
        abort(400)  # missing args
    game = None
    if gameId:  # check gameId not null
        game = Statement.getGame(gameId)
    if not game:  # check gameId null
        if gameName:  # check gameName not null
            game = Statement.findGame(gameName)
    if not game:  # check game null
        abort(404)  # no game found
    lobby = lobbyHandler.findLobbies(game.id)
    lobby = lobby[0] if len(lobby) > 0 else None
    if lobby is not None:
        return jsonify(
            {
                "lobby-id": lobby.id,
                "lobby-addr": lobby.getAddr(),
                "game-id": lobby.gameId,
            }
        )
    else:
        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`.

```python
@main.route("/lobby/create", methods=["POST"])
@auth.login_required
def createLobby():
    gameId = request.json.get("game-id")
    gameName = request.json.get("game-name")
    if not (gameId or gameName):  # check args
        abort(400)  # missing args
    game = None
    if gameId:  # check gameId not null
        game = Statement.getGame(gameId)
    if not game:  # check gameId null
        if gameName:  # check gameName not null
            game = Statement.findGame(gameName)
    if not game:  # check game null
        abort(404)  # no game found
    addr = _getAddr()
    lobby = lobbyHandler.createLobby(addr, game.id)
    return jsonify(
        {"lobby-id": lobby.id, "lobby-addr": lobby.getAddr(), "game-id": lobby.gameId}
    ), 201
```

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

```
@main.route("/lobby/members")
@auth.login_required
def getMembers():
    return jsonify(lobbyHandler.getMembers())

# LobbyHandler.getMembers
def getMembers(self) -> dict[int, list[int]]:
        with self.lobbiesLock:
            return {lobby.id: lobby.members for lobby in self.lobbies}
```

The getFriendLobbies method at /lobby/friends returns all Lobbys 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 Lobbys.

```
@main.route("/lobby/friends")
@auth.login_required
def getFriendLobbies():
    friends = Statement.getFriends(g.account.id)
    lobbyInfo = lambda lobby: {
        "lobby-id": lobby.id,
        "game-id": lobby.gameId,
        "game-name": Statement.getGame(lobby.gameId).name,
    }
    accountInfo = lambda account: {
        "account-id": account.id,
        "username": account.username,
    }
    lobbies = [
        {
            "account": accountInfo(account),
            "lobbies": [
                lobbyInfo(lobby) for lobby in lobbyHandler.getMember(account.id)
            ],
        }
        for account in friends
        if len(lobbyHandler.getMember(account.id)) > 0
    ]
    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()`).

```
def getMember(self, accountId: int) -> list[Lobby]:
    with self.lobbiesLock:
        return [
            lobby
            for lobby in self.lobbies
            if lobby.isNotFull() and accountId in lobby.getMembers()
        ]
```

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`.

```
@main.route("/auth/certificate/validate")
def validateCert():
    valid = False
    certificate = request.json.get("certificate")
    certificate = base64.decodebytes(certificate.encode()) # base64 decode
    if certificate is not None:
        certificate = udp.auth.getCertificateFromDer(certificate) # get x509.Certificate instance
        attributes = udp.auth.getUserCertificateAttributes(certificate)
        if attributes["account-id"] is not None:
            account = Statement.getAccount(attributes["account-id"]) # get Account instance
            publicKey = udp.auth.getRsaPublicFromDer(account.public_key) # get rsa.RSAPublicKey instance
        else:
            publicKey = rsaKey.public_key()
        valid = udp.auth.validateCertificate(certificate, publicKey)
        return jsonify({"valid": valid, "account-id": attributes["account-id"]})
    else:
        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 verity the `certificate`. If a `InvalidSignature Exception` does not arises the method returns `True`. Otherwise, if either the period or verify checks fail, the method returns `False`.

```
def validateCertificate(certificate: x509.Certificate, publicKey: rsa.RSAPublicKey) -> bool:
    # period
    now = datetime.datetime.now(datetime.timezone.utc)
    if not (certificate.not_valid_before_utc <= now <= certificate.not_valid_after_utc): # check in period
        return False
    # signature
    try:
        publicKey.verify( # check against publicKey
            certificate.signature,
            certificate.tbs_certificate_bytes,
            aPadding.PKCS1v15(),
            certificate.signature_hash_algorithm,
        )
    except InvalidSignature:
        return False
    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`.

```
def generateUserCertificate(key, userId: int | str | None = None, username: str | None = None) -> x509.Certificate:
    name = [
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, ORG_NAME),
        x509.NameAttribute(NameOID.COMMON_NAME, COMMON_NAME),
    ]
    if userId is not None:
        name.append(x509.NameAttribute(NameOID.USER_ID, str(userId)))
    if username is not None:
        name.append(x509.NameAttribute(NameOID.PSEUDONYM, username))
    subject = issuer = x509.Name(name)
    cert = (
            ### omitted for clarity
    )
    return cert
```

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

```
def validateCertificate(self, certificate: auth.x509.Certificate) -> bool:
        url = f"http://{self.targetHost}:5000/auth/certificate/validate"
        headers = {"Content-Type": "application/json"}
        certificate = base64.encodebytes(
            auth.getDerFromCertificate(certificate)
        ).decode()
        data = {"certificate": certificate}
        try:
            r = requests.get(url, headers=headers, data=json.dumps(data))
            if r.status_code == 200:
                return r.json()["valid"]
            else:
                return False
        except:
            # server unresponsive
            return False
```

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

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

```
class Choice:
```

```python
    ROCK = 0
    PAPER = 1
    SCISSORS = 2


class Outcome:
    LOOSE = 0
    WIN = 1
    DRAW = 2
```

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

```yaml
NAME: "RPS"
ID: 1
MIN_PLAYERS: 2
MAX_PLAYERS: 2
```

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

```python
# config
CONFIG_PATH = os.path.join(os.path.dirname(__file__), "game_config.yaml")

with open(CONFIG_PATH) as f:
    config = yaml.safe_load(f)

ID = config["ID"]
NAME = config["NAME"]
MIN_PLAYERS = config["MIN_PLAYERS"]
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.

```python
def send(self, addr: tuple[str, int], data: json) -> None:
    self.udpClient.queueDefault(
        addr, flags=lazyFlags(Flag.RELIABLE), data=self.encodeData(data)
    )

def receive(self, addr: tuple[str, int], data: bytes) -> None:
    self.recvQueue.put((addr, self.decodeData(data)))
    if self.onReceiveData:
        self.onReceiveData(addr, data)

@staticmethod
def encodeData(data: dict) -> bytes:
    return json.dumps(data).encode()

@staticmethod
def decodeData(data: bytes) -> dict:
    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.

```python
def gameloop(self) -> None:
    print(f"{bcolors.HEADER}\n\nRock Paper Scissors{bcolors.ENDC}")
    try:
        while self.isRunning:
            choice = None
            print("Choice R[0], P[1], S[2]: ")
            while choice is None:
                try:
                    choice = inputimeout("", timeout=10).strip()
                    if choice == "q":
                        print(
                            f"{bcolors.FAIL}Quitting. Please wait...{bcolors.ENDC}"
                        )
                        self.isRunning = False
                        break
                    choice = int(choice)
                    if choice not in (0, 1, 2):
                        print(
                            f"{bcolors.FAIL}Invalid choice '{choice}'.{bcolors.ENDC}"
                        )
                        choice = None
                except ValueError:
                    print(f"{bcolors.FAIL}Invalid choice.{bcolors.ENDC}")
                    choice = None
                except KeyboardInterrupt:
                    print(f"{bcolors.FAIL}Quitting. Please wait...{bcolors.ENDC}")
                    self.isRunning = False
                    break
                except TimeoutOccurred:
                    if not self.isRunning:
                        break
            if self.isRunning:
                self.send(self.udpClient.targetAddr, {"choice": choice})
                print("Waiting for other player...")
                while self.isRunning:
                    try:
                        addr, data = self.recvQueue.get(timeout=QUEUE_TIMEOUT)
                        break
                    except Empty:
                        pass  # check still running
                if self.isRunning:
                    match data["outcome"]:
                        case 0:
                            o = f"You {bcolors.FAIL}LOOSE{bcolors.ENDC}. "
                        case 1:
                            o = f"You {bcolors.OKGREEN}WIN{bcolors.ENDC}. "
                        case 2:
                            o = f"You {bcolors.OKCYAN}DRAW{bcolors.ENDC}. "
                        case _:
                            o = ""
                    print(
                        f"\n{o}You Picked {data['choice']}. They picked {data['otherChoice']}.\nThe score is {data['score']['score']}:{data['otherScore']['score']}."
                    )
                    if data["outcome"] == Outcome.WIN:
                        self.score += 1
                    self.recvQueue.task_done()
    finally:
        self.udpClient._quit()
```

The `Client` utilizes the `onConnect` to start the `gameThread`.

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

```
                           {error.DisconnectErrorCodes.SERVER_DISCONNECT.name}:
                           {e.args[0]}\nPlease wait while connection closes
                           gracefully...{bcolors.ENDC}"
                    )
                case _:
                    raise e
        if self.gameThread.is_alive():
            self.gameThread.join()
        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.

```
@staticmethod
def evaluateWin(choiceOne: int, choiceTwo: int) -> int:
    match choiceOne:
        case Choice.ROCK:
            match choiceTwo:
                case Choice.ROCK:
                    return Outcome.DRAW
                case Choice.PAPER:
                    return Outcome.LOOSE
                case Choice.SCISSORS:
                    return Outcome.WIN
                case _:
                    raise ValueError
        case Choice.PAPER:
            match choiceTwo:
                case Choice.ROCK:
                    return Outcome.WIN
                case Choice.PAPER:
                    return Outcome.DRAW
                case Choice.SCISSORS:
                    return Outcome.LOOSE
                case _:
                    raise ValueError
        case Choice.SCISSORS:
            match choiceTwo:
                case Choice.ROCK:
                    return Outcome.LOOSE
                case Choice.PAPER:
                    return Outcome.WIN
                case Choice.SCISSORS:
                    return Outcome.DRAW
                case _:
                    raise ValueError
        case _:
            raise ValueError

@staticmethod
def evaluatePlayerChoices(choices: list[tuple[tuple[str, int], int]]):
    outcomes = [
        (choices[0][0], Server.evaluateWin(choices[0][1], choices[1][1])),
        (choices[1][0], Server.evaluateWin(choices[1][1], choices[0][1])),
    ]
    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 `accountIds` and `scores`. These values are retrieved and set through getters and setter which make use of a `thread.Lock` to ensure thread-safety.

```
def playerJoin(self, addr: tuple[str, int], accountId: int) -> None:
    with self.playersLock:
        self.players[addr] = {"score": 0, "accountId": accountId}
    if self.onClientJoin:
        self.onClientJoin(addr, accountId)

def playerLeave(self, addr: tuple[str, int], accountId: int) -> None:
    with self.playersLock:
        del self.players[addr]
    if self.onClientLeave:
        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.

```
def mainloop(self) -> None:
        self.udpServer.startThreads()
        try:
            while self.isRunning:
                if self.playerCount == MAX_PLAYERS:
                    choices = self.getChoices()
                    outcomes = self.evaluatePlayerChoices(choices)
                    replies = {}
                    for addr, outcome in outcomes:
                        replies[addr] = {
                            "outcome": outcome,
                            "choice": [v for k, v in choices if k == addr][0],
                            "otherChoice": [v for k, v in choices if k != addr][0],
                        }
                        if outcome == Outcome.WIN:
                            self.incrementPlayer(addr)
                    scores = self.getPlayers()
                    for addr in replies:
                        replies[addr] |= {
                            "score": scores[addr],
                            "otherScore": [v for k, v in scores.items() if k != addr][
                                0
                            ],
                        }
                        self.send(addr, replies[addr])
        finally:
            self.quit()
```

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

```
def getChoices(self) -> list[tuple[tuple[str, int], int]]:
        choices = {}
        while self.isRunning:
            try:
                addr, data = self.recvQueue.get(timeout=QUEUE_TIMEOUT)
                choices[addr] = data["choice"]
                if len(choices) == 2:
                    choices = [(addr, choice) for addr, choice in choices.items()]
                    self.recvQueue.task_done()
                    return choices
            except Empty:
                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.

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

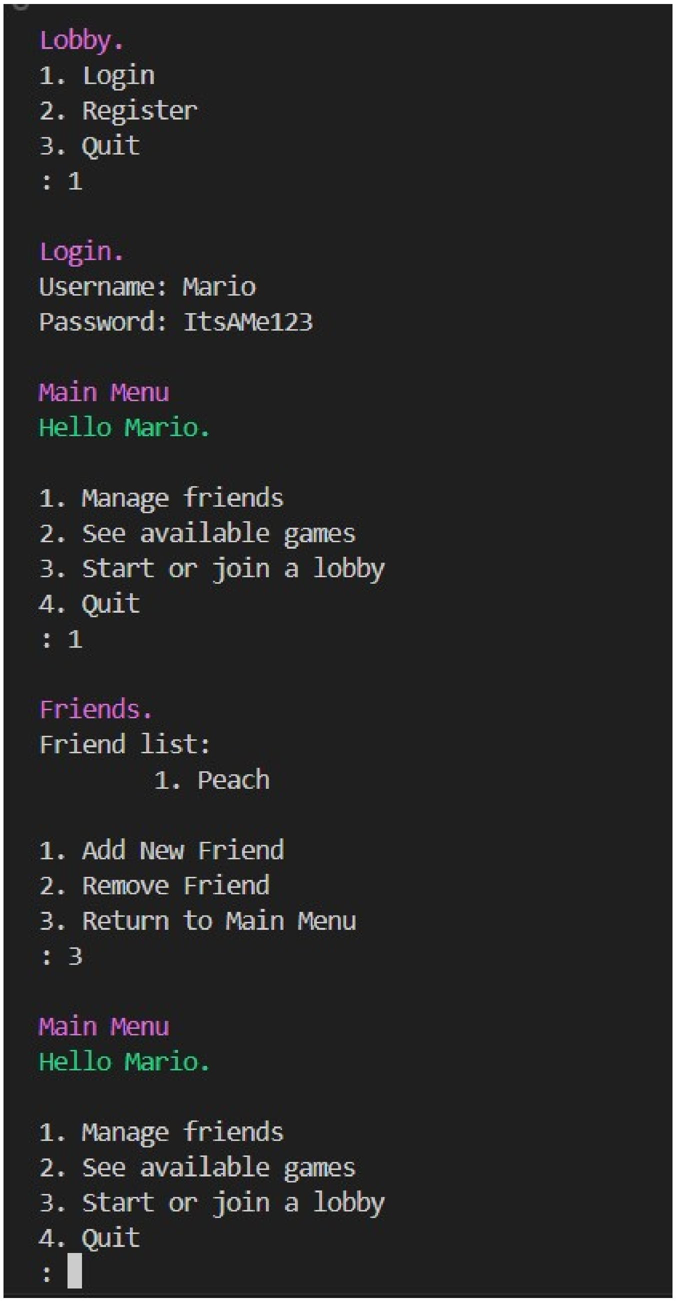
The Client then uses requests.auth.HTTPBasicAuth for the rest of the session communications.

```python
class Client:
    def __init__(self, username: str, password: str, token: str | None = None) -> None:
        self.username = username
        self.password = password
        self.gameClient = None
        self.token = (
            token if token is not None else self.getToken(self.username, self.password)
        )
        self.auth = HTTPBasicAuth(self.token, "")
        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.

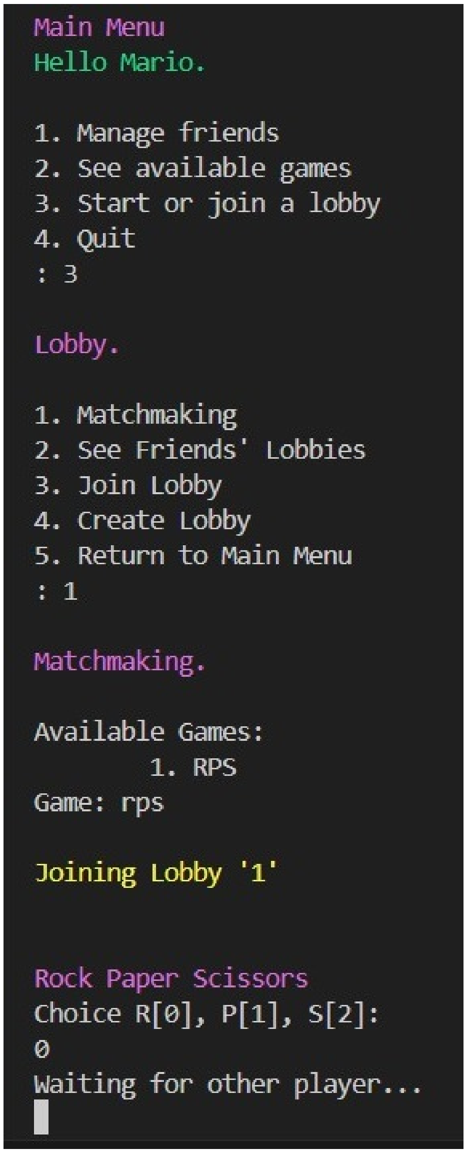


```
Lobby.
1. Login
2. Register
3. Quit
: 1

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`.

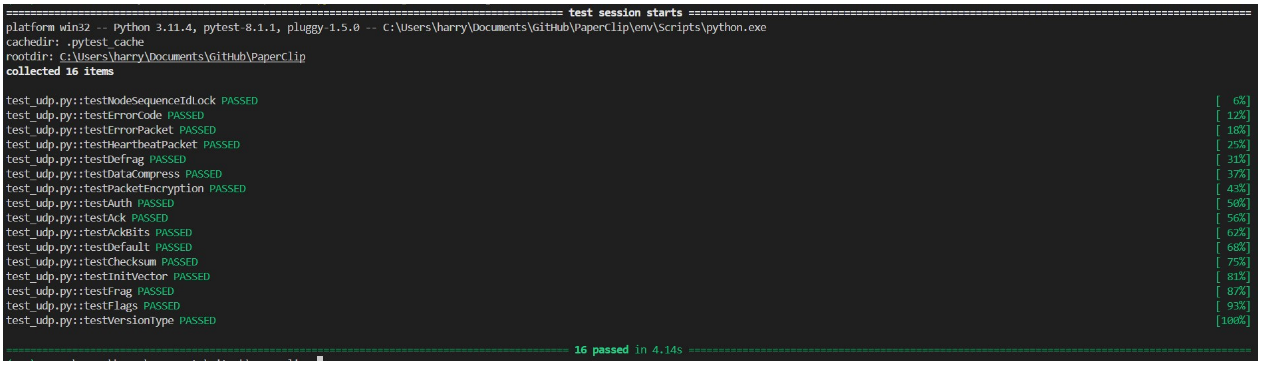
```python
def _matchmaking(client: Client):
    print(f"{bcolors.HEADER}\nMatchmaking.{bcolors.ENDC}")
    game = _gameInput(client)
    if game is None:
        return None
    try:
        lobby = client.findLobby(gameName=game)
    except AssertionError:
        lobby = client.createLobby(gameName=game)
    time.sleep(1)
    client.join(lobby["lobby-id"])
    return None
```

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

```python
def join(self, lobbyId: int) -> None:
        print(f"\n{bcolors.WARNING}Joining Lobby '{lobbyId}'{bcolors.ENDC}")
        data = self.getLobby(lobbyId)
        if data["lobby-addr"] is not None:
            match data["game-id"]:
                case 1:
                    self.gameClient = RpsClient(
                        (TCP_HOST, C_PORT),
                        data["lobby-addr"],
                        rsaKey=self.key,
                        userId=self.id,
                        username=self.username,
                    )
                    self.gameClient.connect()
                case _:
                    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::testAckBits 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 `Lobby`s.

The `client` package provides a user-friendly interface for interactions with the `Server`. Additionally, using the information provided by the `Server`, is able to matchmake 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 subtile 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`. `ECHDE` 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): JTW 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) 'Inputimeout', *GitHub repository*. GitHub. Available at: https://github.com/johejo/inputimeout.

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-sever.

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.

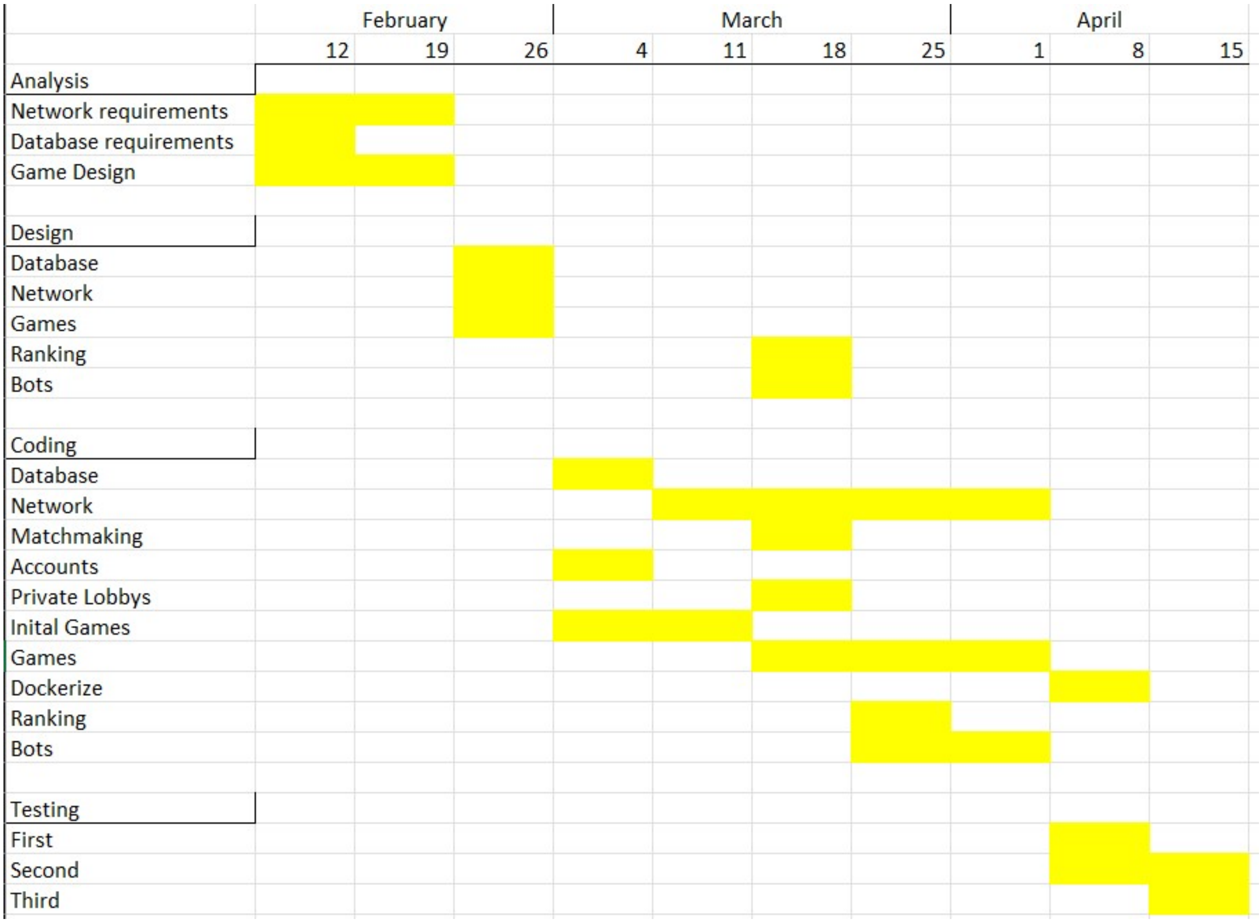
Further objectives include:

- 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.



	February			March				April		
	12	19	26	4	11	18	25	1	8	15
Analysis										
Network requirements	■	■								
Database requirements	■									
Game Design	■	■								
Design										
Database			■							
Network			■							
Games			■							
Ranking						■				
Bots						■				
Coding										
Database				■						
Network					■	■	■	■		
Matchmaking						■				
Accounts				■						
Private Lobbys						■				
Inital Games				■	■					
Games						■	■	■		
Dockerize									■	
Ranking							■			
Bots							■	■		
Testing										
First									■	
Second									■	■
Third										■

Figure 9: workplan

Risks

Objective	Risk	Severity	Score	Risks	Actions
Network	2	5	10	Vulnerabilities	Ensure that the system is not vulnerable to any malicious interaction. Including ‘cheating’.
Accounts	1	4	4	GDPR	Limit type of stored data and where data is relevant unsure stored in accordance to GDPR.
Safeguarding	2	4	8	User safety	Limit interactions between user accounts.
Development	5	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	3	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`.

```
# docker-compose.yaml
version: '3.1'

services:

  adminer:
    image: adminer
    ports:
      - 8080:8080

  db:
    image: mysql:5.6
    environment:
      MYSQL_ROOT_PASSWORD: root
    ports:
      - 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

```
# udp
S_HOST=127.0.0.1
S_PORT=2024
C_HOST=127.0.0.1
C_PORT=2025
## node
SOCKET_BUFFER_SIZE = 1024
SEND_SLEEP_TIME = 0.1
QUEUE_TIMEOUT = 10
SOCKET_TIMEOUT = 20
## server
HEARTBEAT_MAX_TIME = 120
HEARTBEAT_MIN_TIME = 30
MAX_CLIENTS
## auth
```

```
ORG_NAME = Paperclip
COMMON_NAME = 127.0.0.1
## utils
MAX_FRAGMENT_SIZE = 988

# client
TCP_PORT = 5000

# app
FLASK_APP=server
PRUNE_TIME = 58
SECRET_KEY = MyVerySecretKey
SQLALCHEMY_DATABASE_URI = mysql://root:root@localhost:3306/paperclip

# debug
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

```python
# udp.__init__
import logging
import os
import sys

import dotenv

__version__ = 0

dotenv.load_dotenv(".env")
S_HOST = os.environ.get("S_HOST")
S_PORT = int(os.environ.get("S_PORT"))
C_HOST = os.environ.get("C_HOST")
C_PORT = int(os.environ.get("C_PORT"))
# node
SOCKET_BUFFER_SIZE = int(os.environ.get("SOCKET_BUFFER_SIZE"))
```

```
SEND_SLEEP_TIME = float(os.environ.get("SEND_SLEEP_TIME"))
QUEUE_TIMEOUT = int(os.environ.get("QUEUE_TIMEOUT"))
SOCKET_TIMEOUT = int(os.environ.get("SOCKET_TIMEOUT"))
# server
HEARTBEAT_MAX_TIME = int(os.environ.get("HEARTBEAT_MAX_TIME"))
HEARTBEAT_MIN_TIME = int(os.environ.get("HEARTBEAT_MIN_TIME"))
MAX_CLIENTS = (
    int(os.environ.get("MAX_CLIENTS"))
    if os.environ.get("MAX_CLIENTS") is not None
    else float("inf")
)
# auth
ORG_NAME = os.environ.get("ORG_NAME")
COMMON_NAME = os.environ.get("COMMON_NAME")
# utils
MAX_FRAGMENT_SIZE = int(os.environ.get("MAX_FRAGMENT_SIZE"))


class bcolors:
    HEADER = "\033[95m"
    OKBLUE = "\033[94m"
    OKCYAN = "\033[96m"
    OKGREEN = "\033[92m"
    WARNING = "\033[93m"
    FAIL = "\033[91m"
    ENDC = "\033[0m"
    BOLD = "\033[1m"
    UNDERLINE = "\033[4m"


class ColorFilter(logging.Filter):
    colorCodes = [
        getattr(bcolors, attr) for attr in dir(bcolors) if not
            attr.startswith("__")
    ]

    def filter(self, record: logging.LogRecord) -> bool:
        for color in self.colorCodes:
            record.msg = record.msg.replace(color, "")
        return True


logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

```

```python
printHandler = logging.StreamHandler(sys.stdout)
printHandler.setLevel(logging.INFO)
printHandler.setFormatter(
    logging.Formatter(f"{bcolors.OKBLUE}%(threadName)s{bcolors.ENDC} - %(message)s")
)
logger.addHandler(printHandler)

fileHandler = logging.FileHandler("paperclip.log")
fileHandler.setLevel(logging.DEBUG)
fileHandler.addFilter(ColorFilter())
fileHandler.setFormatter(
    logging.Formatter("%(asctime)s - %(levelname)s - %(threadName)s - %(message)s")
)
logger.addHandler(fileHandler)
```

```python
# upd.__main__
from . import client, server


def runServer():
    s = server.Server((S_HOST, S_PORT))
    s.startThreads()
    return s


def runClient():
    c = client.Client((C_HOST, C_PORT), (S_HOST, S_PORT))
    c.connect()
    return c


if __name__ == "__main__":
    import time

    from . import C_HOST, C_PORT, S_HOST, S_PORT

    s = runServer()
    time.sleep(1)
    c = runClient()
    time.sleep(1)
    x = None
    x = input("> ")
    while x != "END":
        c.queueDefault(data=x.encode())
        x = input("> ")
    c.isRunning.clear()
    time.sleep(1)
    s.isRunning.clear()
    time.sleep(1)
    print("END")
```

```
# udp.auth
import datetime
import os

from cryptography import x509
from cryptography.exceptions import InvalidSignature
from cryptography.hazmat.primitives import hashes, hmac, padding,
   serialization
from cryptography.hazmat.primitives.asymmetric import ec, rsa
from cryptography.hazmat.primitives.asymmetric import padding as aPadding
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes
from cryptography.hazmat.primitives.kdf.hkdf import HKDF
from cryptography.x509.oid import NameOID

from . import COMMON_NAME, ORG_NAME


def generateRsaKey() -> rsa.RSAPrivateKey:
    key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
    )
    return key


def getDerFromRsaPrivate(key: rsa.RSAPrivateKey, password: bytes) -> bytes:
    der = key.private_bytes(
        encoding=serialization.Encoding.DER,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.BestAvailableEncryption(password),
    )
    return der

```

```python

def getRsaPrivateFromDer(data: bytes, password: bytes) -> rsa.RSAPrivateKey:
    key = serialization.load_der_private_key(data, password=password)
    return key


def getDerFromRsaPublic(key: rsa.RSAPublicKey) -> bytes:
    der = key.public_bytes(
        encoding=serialization.Encoding.DER,
        format=serialization.PublicFormat.SubjectPublicKeyInfo,
    )
    return der


def getRsaPublicFromDer(data: bytes) -> rsa.RSAPublicKey:
    key = serialization.load_der_public_key(data)
    return key


def generateUserCertificate(
    key, userId: int | str | None = None, username: str | None = None
) -> x509.Certificate:
    name = [
        x509.NameAttribute(NameOID.ORGANIZATION_NAME, ORG_NAME),
        x509.NameAttribute(NameOID.COMMON_NAME, COMMON_NAME),
    ]
    if userId is not None:
        name.append(x509.NameAttribute(NameOID.USER_ID, str(userId)))
    if username is not None:
        name.append(x509.NameAttribute(NameOID.PSEUDONYM, username))
    subject = issuer = x509.Name(name)
    cert = (
        x509.CertificateBuilder()
        .subject_name(subject)
        .issuer_name(issuer)
        .public_key(key.public_key())
        .serial_number(x509.random_serial_number())
        .not_valid_before(datetime.datetime.now(datetime.timezone.utc))
        .not_valid_after(
            datetime.datetime.now(datetime.timezone.utc) +
                datetime.timedelta(days=1)
        )
        .add_extension(
            x509.SubjectAlternativeName([x509.DNSName("localhost")]),
            critical=False,
        )
        .sign(key, hashes.SHA256())
    )

    return cert


def getUserCertificateAttributes(certificate: x509.Certificate) -> list:
    accountId = certificate.subject.get_attributes_for_oid(NameOID.USER_ID)
    accountId = accountId[0].value if len(accountId) > 0 else None
    username = certificate.subject.get_attributes_for_oid(NameOID.PSEUDONYM)
    username = username[0].value if len(username) > 0 else None
    return {"account-id": accountId, "username": username}


def validateCertificate(
    certificate: x509.Certificate, publicKey: rsa.RSAPublicKey
) -> bool:
    # period
    now = datetime.datetime.now(datetime.timezone.utc)
    if not (certificate.not_valid_before_utc <= now <= certificate.not_valid_after_utc):
        return False
    # signature
    try:
        publicKey.verify(
            certificate.signature,
            certificate.tbs_certificate_bytes,
            aPadding.PKCS1v15(),
            certificate.signature_hash_algorithm,
        )
    except InvalidSignature:
        return False
    return True


def generateEcKey() -> ec.EllipticCurvePrivateKey:
    key = ec.generate_private_key(ec.SECP384R1())
    return key


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


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


def getDerFromCertificate(certificate: x509.Certificate) -> bytes:
    return certificate.public_bytes(serialization.Encoding.DER)


def getCertificateFromDer(certificateDer: bytes) -> x509.Certificate:
    return x509.load_der_x509_certificate(certificateDer)


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


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


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


def generateInitVector() -> bytes:
    return os.urandom(16)


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


def generateFinished(sessionKey: bytes, finishedLabel: bytes, messages: bytes):
    hashValue = hashes.Hash(hashes.SHA256())
    hashValue.update(messages)
    hashValue = hashValue.finalize()

    prf = hmac.HMAC(sessionKey, hashes.SHA256())
    prf.update(finishedLabel)
    prf.update(hashValue)
    prf = prf.finalize()

    return prf
```

```python
# udp.client
import base64
import json
import socket

import requests
from cryptography.hazmat.primitives.asymmetric.rsa import RSAPrivateKey

from udp.error import Major, Minor

from . import auth, bcolors, error, logger, node, packet


class Client(node.Node):
    targetAddr: tuple[str, int]
    rsaKey: RSAPrivateKey
    onConnect: None
    onDisconnect: None

    def __init__(
        self,
        addr,
        targetAddr,
        rsaKey: RSAPrivateKey | None = None,
        accountId: int | str | None = None,
        username: str | None = None,
        onConnect=None,
        onDisconnect=None,
        onReceiveData=None,
    ) -> None:
        self.targetAddr = targetAddr
        self.rsaKey = rsaKey if rsaKey is not None else auth.generateRsaKey()
        self.onConnect = onConnect
        self.onDisconnect = onDisconnect
        s = socket.socket(type=socket.SOCK_DGRAM)
        super().__init__(
            addr,
            cert=auth.generateUserCertificate(self.rsaKey, accountId, username),
            accountId=accountId,
            socket=s,
            onReceiveData=onReceiveData,
        )
        self.regenerateEcKey()
        self.bind(self.addr)

    @property
    def targetHost(self) -> str | None:
        return self.targetAddr[0] if self.targetAddr is not None else None

    @property
    def targetPort(self) -> int | None:
        return self.targetAddr[1] if self.targetAddr is not None else None

    def queueDefault(
        self,
        addr: tuple[str, int] = None,
        flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
        data: bytes | None = None,
    ) -> None:
        return super().queueDefault(self.targetAddr, flags=flags, data=data)

    def queueACK(
        self,
        addr: tuple[str, int] = None,
        ackId: int = None,
        flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
        data: bytes | None = None,
    ) -> None:
        return super().queueACK(self.targetAddr, ackId, flags=flags, data=data)

    def queueError(
        self,
        addr: tuple[str, int] = None,
        major: Major | int = 0,
        minor: Minor | int = 0,
        flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
        data: bytes | None = None,
    ) -> None:
        return super().queueError(self.targetAddr, major, minor, flags, data)

    def queueDisconnect(
        self,
        addr: tuple[str, int] = None,
        flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
        data: bytes | None = None,
    ) -> None:
        self.queueError(
            self.targetAddr,
            flags=flags,
            major=error.Major.DISCONNECT,
            minor=error.DisconnectErrorCodes.CLIENT_DISCONNECT,
            data=data,
        )

    def connect(self) -> None:
        try:
            logger.info(
                f"{bcolors.WARNING}# Handshake with {self.targetAddr} starting.{bcolors.ENDC}"
            )
            self.outboundThread.start()
            self.queueAuth(self.targetAddr, self.cert, self.ecKey.public_key())
            authPacket = None
            ackPacket = None
            while True:
                p, addr = self.receivePacket()
                if p is not None:
                    # logic
                    if p.packet_type == packet.Type.AUTH:
                        logger.info(
                            f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC} {bcolors.OKCYAN}{p}{bcolors.ENDC}"
                        )
                        authPacket = p
                        self.sessionKey = auth.generateSessionKey(
                            self.ecKey, p.public_key
                        )
                        if not self.validateCertificate(p.certificate):
                            logger.critical(f"Invalid peer cert {p.certificate}")
                            self.queueError(
                                major=error.Major.CONNECTION,
                                minor=error.ConnectionErrorCodes.CERTIFICATE_INVALID,
                                data=b"Invalid Certificate.",
                            )
                            break
                        self.queueFinished(
                            self.targetAddr, p.sequence_id, self.sessionKey
                        )
                    elif p.packet_type == packet.Type.ACK:
                        logger.info(
                            f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC} {bcolors.OKCYAN}{p}{bcolors.ENDC}"
                        )
                        ackPacket = p
                        self.receiveAck(p, addr)
                    elif p.packet_type == packet.Type.ERROR:
                        logger.info(
                            f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC} {bcolors.OKCYAN}{p}{bcolors.ENDC}"
                        )
                        self.receive(p, addr)
                    else:
                        logger.warning(
                            f"{bcolors.WARNING}! {addr} :{bcolors.ENDC} {bcolors.WARNING}{p}{bcolors.ENDC}"
                        )
                    if authPacket is not None and ackPacket is not None:
```

```python
                        break
                else:
                    # timeout and abort
                    logger.critical("Server not responsive.")
            if self.validateHandshake(p.data):
                # success
                logger.info(
                    f"{bcolors.OKGREEN}Handshake success starting mainloop...{bcolors.ENDC}"
                )
                self.inboundThread.start()
                if self.onConnect:
                    self.onConnect(addr)
            else:
                logger.critical(
                    f"Local finished value {node.Node._generateFinished(self.sessionKey)} does not match peer finished value {ackPacket.data}"
                )
                self.queueError(
                    major=error.Major.CONNECTION,
                    minor=error.ConnectionErrorCodes.FINISH_INVALID,
                    data=b"Invalid finish.",
                )
        except error.PaperClipError as e:
            raise e
        except Exception as e:
            raise e
        else:
            if self.isRunning.is_set():
                self._quit()

    # auth
    def validateCertificate(self, certificate: auth.x509.Certificate) -> bool:
        url = f"http://{self.targetHost}:5000/auth/certificate/validate"
        headers = {"Content-Type": "application/json"}
        certificate = base64.encodebytes(
            auth.getDerFromCertificate(certificate)
        ).decode()
        data = {"certificate": certificate}
        r = requests.get(url, headers=headers, data=json.dumps(data))
        if r.status_code == 200:
            return r.json()["valid"]
        else:
            return False

    # misc
    def quit(self, msg: str = "quit call", e: Exception = None) -> None:
        self.queueDisconnect(data=msg.encode())
        self.queue.join()
        super().quit(msg, e)

    def handleDisconnectError(
        self, p: packet.ErrorPacket, addr: tuple[str, int], e: error.DisconnectError
    ) -> None:
        match e:
            case error.ServerDisconnectError():
                self._quit(e)
            case error.ClientDisconnectError():
                pass  # should not react to client disconnect
            case _:
                raise e

    def mainloop(self, onQuit=None) -> None:
        try:
            while self.isRunning.is_set():
                pass
        except KeyboardInterrupt:
            print(f"{bcolors.FAIL}Quitting. Please wait...{bcolors.ENDC}")
        finally:
            if onQuit is None:
                self.quit(e=self.exitError)
            else:
                onQuit(e=self.exitError)
```

```python
# udp.error
from enum import Enum


class Major(Enum):
    ERROR = 0
    CONNECTION = 1
    DISCONNECT = 2
    PACKET = 3


class Minor(Enum):
    pass


class PaperClipError(Exception):
    """Unknown error"""


# connection
class ConnectionErrorCodes(Minor):
    CONNECTION = 0
    NO_SPACE = 1
    CERTIFICATE_INVALID = 2
    FINISH_INVALID = 3


class ConnectionError(PaperClipError):
    """Handshake connection could not be finished"""


class NoSpaceError(ConnectionError):
    """Server has insufficient space to accept new clients"""


class CertificateInvalidError(ConnectionError):
    """Certificate is invalid / can not be validated"""


class FinishInvalidError(ConnectionError):
    """Finish is invalid"""


_connectionErrors = {
    ConnectionErrorCodes.CONNECTION: ConnectionError,
    ConnectionErrorCodes.NO_SPACE: NoSpaceError,
    ConnectionErrorCodes.CERTIFICATE_INVALID: CertificateInvalidError,
    ConnectionErrorCodes.FINISH_INVALID: FinishInvalidError,
}


def getConnectionError(minor: ConnectionErrorCodes | int) -> ConnectionError:
    try:
        minor = minor if isinstance(minor, Minor) else ConnectionErrorCodes(minor)
        if minor in _connectionErrors:
            return _connectionErrors[minor]
        else:
            return PaperClipError
    except ValueError:
        return PaperClipError


def getConnectionCode(error: ConnectionError) -> ConnectionErrorCodes:
    try:
        return list(_connectionErrors.keys())[
            list(_connectionErrors.values()).index(error)
        ]
    except ValueError:
        return PaperClipError


# disconnect
class DisconnectErrorCodes(Minor):
    DISCONNECT = 0
    SERVER_DISCONNECT = 1
    CLIENT_DISCONNECT = 2


class DisconnectError(PaperClipError):
    """A party is disconnecting"""


class ServerDisconnectError(DisconnectError):
    """The server is closing"""


class ClientDisconnectError(DisconnectError):
    """The client is closing"""


_disconnectErrors = {
    DisconnectErrorCodes.DISCONNECT: DisconnectError,
    DisconnectErrorCodes.SERVER_DISCONNECT: ServerDisconnectError,
    DisconnectErrorCodes.CLIENT_DISCONNECT: ClientDisconnectError,
}


def getDisconnectError(minor: DisconnectErrorCodes | int) -> DisconnectError:
    try:
        minor = minor if isinstance(minor, Minor) else
           DisconnectErrorCodes(minor)
```

```python
        if minor in _disconnectErrors:
            return _disconnectErrors[minor]
        else:
            return PaperClipError
    except ValueError:
        return PaperClipError


def getDisconnectCode(error: DisconnectError) -> DisconnectErrorCodes:
    try:
        return list(_disconnectErrors.keys())[
            list(_disconnectErrors.values()).index(error)
        ]
    except ValueError:
        return PaperClipError


# packet
class PacketErrorCodes(Minor):
    PACKET = 0
    VERSION = 1
    PACKET_TYPE = 2
    FLAGS = 3
    SEQUENCE_ID = 4
    FRAGMENT_ID = 5
    FRAGMENT_NUMBER = 6
    INIT_VECTOR = 7
    COMPRESSION = 8
    CHECKSUM = 9


class PacketError(PaperClipError):
    """Packet cannot be read"""


class VersionError(PacketError):
    """Packet Version is invalid / does not match expected"""


class PacketTypeError(PacketError):
    """Packet Type is invalid / unknown"""


class FlagsError(PacketError):
    """Flags are invalid / unknown"""
```

```python


class SequenceIdError(PacketError):
    """Sequence Id is invalid / does not match expected"""


class FragmentIdError(PacketError):
    """Fragment Id is invalid / unknown"""


class FragmentNumberError(PacketError):
    """Fragment Number is invalid / unknown"""


class InitVectorError(PacketError):
    """Init Vector is invalid / unknown i.e. decrypt fail"""


class CompressionError(PacketError):
    """Decompression fail"""


class ChecksumError(PacketError):
    """Checksum is invalid / unknown i.e. checksum fail"""


_packetErrors = {
    PacketErrorCodes.PACKET: PacketError,
    PacketErrorCodes.VERSION: VersionError,
    PacketErrorCodes.PACKET_TYPE: PacketTypeError,
    PacketErrorCodes.FLAGS: FlagsError,
    PacketErrorCodes.SEQUENCE_ID: SequenceIdError,
    PacketErrorCodes.FRAGMENT_ID: FragmentIdError,
    PacketErrorCodes.FRAGMENT_NUMBER: FragmentNumberError,
    PacketErrorCodes.INIT_VECTOR: InitVectorError,
    PacketErrorCodes.COMPRESSION: CompressionError,
    PacketErrorCodes.CHECKSUM: ChecksumError,
}


def getPacketError(minor: PacketErrorCodes | int) -> PacketError:
    try:
        minor = minor if isinstance(minor, Minor) else PacketErrorCodes(minor)
        if minor in _packetErrors:
            return _packetErrors[minor]
        else:
            return PaperClipError
    except ValueError:
        return PaperClipError


def getPacketCode(error: PacketError) -> PacketErrorCodes:
    try:
        return
           list(_packetErrors.keys())[list(_packetErrors.values()).index(error)]
    except ValueError:
        return PaperClipError


# convenience
def getError(major: Major | int, minor: Minor | int = 0) -> PaperClipError:
    try:
        major = major if isinstance(major, Major) else Major(major)
        match major:
            case Major.CONNECTION:
                return getConnectionError(minor)
            case Major.DISCONNECT:
                return getDisconnectError(minor)
            case Major.PACKET:
                return getPacketError(minor)
            case _:
                return PaperClipError
    except TypeError:
        return PaperClipError


def getMinor(major: Major, minor: int) -> Minor:
    match major:
        case Major.CONNECTION:
            return ConnectionErrorCodes(minor)
        case Major.DISCONNECT:
            return DisconnectErrorCodes(minor)
        case Major.PACKET:
            return PacketErrorCodes(minor)
        case _:
            return Minor


def getErrorCode(error: PaperClipError) -> tuple[Major, Minor]:
    match error:
        case c if issubclass(c, ConnectionError):
            return (Major.CONNECTION, getConnectionCode(error))
        case d if issubclass(d, DisconnectError):
            return (Major.DISCONNECT, getDisconnectCode(error))
        case p if issubclass(p, PacketError):
            return (Major.PACKET, getPacketCode(error))
        case _:
            return (Major.ERROR, Minor)
```

```
# udp.node
import time
from datetime import datetime
from queue import Empty, Queue
from socket import SOCK_DGRAM
from socket import socket as Socket
from threading import Event, Lock, Thread, get_ident

from cryptography.hazmat.primitives.asymmetric.ec import EllipticCurvePrivateKey
from cryptography.x509 import Certificate

from . import (
    QUEUE_TIMEOUT,
    SEND_SLEEP_TIME,
    SOCKET_BUFFER_SIZE,
    SOCKET_TIMEOUT,
    auth,
    bcolors,
    error,
    logger,
    packet,
)

ACK_RESET_SIZE = (2**packet.ACK_BITS_SIZE) // 2


class Node:
    addr: tuple[str, int]
    _sequenceId: int
    sentAckBits = list[bool | None]
    recvAckBits = list[bool | None]
    newestSeqId: int | None
    fragBuffer: dict[int, list[packet.Packet]]
```

```python
    queue: Queue
    heartbeat: datetime | None
    # id
    cert: Certificate | None
    _accountId: int | None
    # session
    ecKey: EllipticCurvePrivateKey
    sessionKey: bytes | None
    handshake: bool
    # threads
    inboundThread: Thread
    outboundThread: Thread
    sequenceIdLock: Lock
    sendLock: Lock
    isRunning: Event
    # socket
    socket: Socket | None
    # callback
    onReceiveData: None
    # exitCode
    exitError: error.PaperClipError | None

    def __init__(
        self,
        addr: tuple[str, int],
        cert: Certificate | None = None,
        accountId: int | None = None,
        sendLock: Lock = Lock(),
        socket: Socket | None = Socket(type=SOCK_DGRAM),
        onReceiveData: None = None,
    ) -> None:
        self.addr = addr
        self.sequenceId = 0
        self.sentAckBits = [None for _ in range(2**packet.ACK_BITS_SIZE)]
        self.recvAckBits = [None for _ in range(2**packet.ACK_BITS_SIZE)]
        self.newestSeqId = 0
        self.fragBuffer = {}
        self.queue = Queue()
        # id
        self.cert = cert
        self.accountId = accountId
        # session
        self.sessionKey = None
        self.handshake = False
        # threads
        self.inboundThread = Thread(
            name=f"{self.port}:Inbound", target=self.listen, daemon=True
        )
        self.outboundThread = Thread(
            name=f"{self.port}:Outbound", target=self.sendQueue, daemon=True
        )
        self.sequenceIdLock = Lock()
        self.sendLock = sendLock
        self.isRunning = Event()
        self.isRunning.set()
        # socket
        self.socket = socket
        self.socket.settimeout(SOCKET_TIMEOUT)
        # callback
        self.onReceiveData = onReceiveData
        # exit
        self.exitError = None

    def bind(self, addr):
        self.socket.bind(addr)

    # properties
    @property
    def host(self) -> str:
        return self.addr[0]

    @property
    def port(self) -> int:
        return self.addr[1]

    @property
    def sequenceId(self) -> int:
        return self._sequenceId

    @sequenceId.setter
    def sequenceId(self, v: int) -> None:
        self._sequenceId = v % 2**packet.SEQUENCE_ID_SIZE

    def incrementSequenceId(self, addr: tuple[str, int]) -> None:
        with self.getSequenceIdLock(addr):
            self.sequenceId += 1

    @property
    def accountId(self) -> int:
        return self._accountId

    @accountId.setter
    def accountId(self, v: int | str | None) -> None:
        try:
            self._accountId = int(v)
        except ValueError:
            self._accountId = v
        except TypeError:
            self._accountId = None

    def getSentAckBit(self, addr: tuple[str, int], p: packet.Packet) -> bool | None:
        return self.sentAckBits[p.sequence_id]

    def setSentAckBit(self, addr: tuple[str, int], ackBit: int, v: bool) -> None:
        self.sentAckBits[ackBit] = v

    def getSentAckBits(self, addr: tuple[str, int]) -> list[bool | None]:
        return self.sentAckBits

    def getRecvAckBit(self, addr: tuple[str, int], p: packet.Packet) -> bool | None:
        return self.recvAckBits[p.sequence_id]

    def getRecvAckBits(self, addr: tuple[str, int]) -> list[bool | None]:
        return self.recvAckBits

    def setRecvAckBit(self, addr: tuple[str, int], ackBit: int, v: bool) -> None:
        self.recvAckBits[ackBit] = v

    def resetRecvAckBits(self, addr: tuple[str, int]) -> None:
        recvAckBits = self.getRecvAckBits(addr)
        newestSeqId = self.getNewestSeqId(addr)
        pointer = (newestSeqId - ACK_RESET_SIZE) % 2**packet.ACK_BITS_SIZE
        counter = 0
        while counter != pointer:
            recvAckBits[(newestSeqId + 1 + counter) % 2**packet.ACK_BITS_SIZE] = None
            counter += 1

    def getNewestSeqId(self, addr: tuple[str, int]) -> int:
        return self.newestSeqId

    def setNewestSeqId(self, addr: tuple[str, int], newestSeqId: int) -> None:
        self.newestSeqId = newestSeqId

    @staticmethod
    def getNewerSeqId(currentSeqId: int, newSeqId: int) -> int:
        currentDiff = (newSeqId - currentSeqId) % (2**packet.SEQUENCE_ID_SIZE)
        newDiff = (currentSeqId - newSeqId) % (2**packet.SEQUENCE_ID_SIZE)
        if newDiff < currentDiff:
            return currentSeqId
        else:
            return newSeqId

    def getSessionKey(self, addr: tuple[str, int]) -> int:
        return self.sessionKey

    def getHandshake(self, addr: tuple[str, int]) -> bool:
        return self.handshake

    def getFragBuffer(self, addr: tuple[str, int]) -> dict[int,
       list[packet.Packet]]:
        return self.fragBuffer

    def getSequenceId(self, addr: tuple[str, int]) -> int:
        return self.sequenceId

    def getSequenceIdLock(self, addr: tuple[str, int]) -> Lock:
        return self.sequenceIdLock

    def getQueue(self, addr: tuple[str, int]) -> Queue:
        return self.queue

    def getHeartbeat(self, addr: tuple[str, int]) -> datetime:
        return self.heartbeat

    def setHeartbeat(self, addr: tuple[str, int], v: datetime) -> None:
        self.heartbeat = v

    def regenerateEcKey(self) -> None:
        self.ecKey = auth.generateEcKey()

    # sends
    def sendPacket(self, addr: tuple[str, int], p: packet.Packet) -> None:
        with self.sendLock:
            try:
                self.socket.sendto(p.pack(p), (addr[0], addr[1]))
                logger.info(
                    f"{bcolors.OKBLUE}> {addr} :{bcolors.ENDC}
                       {bcolors.OKCYAN}{p}{bcolors.ENDC}"
                )
            except error.PacketError as e:
                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}"
                )

    def sendQueue(self) -> None:
        while self.isRunning.is_set():
            try:
                addr, p = self.queue.get(timeout=QUEUE_TIMEOUT)
                if p.flags[packet.Flag.RELIABLE.value]:
                    if self.getSentAckBit(addr, p):
                        self.queue.task_done()
                        continue
                    else:
                        self.sendPacket(addr, p)
                        self.queue.task_done()
                        self.queue.put((addr, p))
                else:
                    self.sendPacket(addr, p)
                    self.queue.task_done()
                time.sleep(SEND_SLEEP_TIME)
            except Empty:
                pass  # check still running
        else:
            logger.info("| sendQueue thread stopping...")

    def queuePacket(self, addr: tuple[str, int], p: packet.Packet) -> None:
        if p.flags[packet.Flag.RELIABLE.value]:
            self.setSentAckBit(addr, p.sequence_id, False)
        if p.flags[packet.Flag.CHECKSUM.value]:
            p.setChecksum()
        if p.flags[packet.Flag.COMPRESSED.value]:
            p.compressData()
        if p.flags[packet.Flag.ENCRYPTED.value]:
            p.encryptData(self.getSessionKey(addr))
        if p.flags[packet.Flag.FRAG.value]:
            frags = p.fragment()
            for frag in frags:
```

```python
                self.getQueue(addr).put((addr, frag))
        else:
            self.getQueue(addr).put((addr, p))

    def queueDefault(
        self,
        addr: tuple[str, int],
        flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
        data: bytes | None = None,
    ) -> None:
        p = packet.Packet(sequence_id=self.getSequenceId(addr), flags=flags, data=data)
        self.incrementSequenceId(addr)
        self.queuePacket(addr, p)

    def queueACK(
        self,
        addr: tuple[str, int],
        ackId: int,
        flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
        data: bytes | None = None,
    ) -> None:
        ack_bits = self.packRecvAckBits(self.getRecvAckBits(addr), ackId)
        p = packet.AckPacket(
            sequence_id=self.getSequenceId(addr),
            flags=flags,
            ack_id=ackId,
            ack_bits=ack_bits,
            data=data,
        )
        self.incrementSequenceId(addr)
        self.queuePacket(addr, p)

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

    def queueFinished(
        self, addr: tuple[str, int], seqId: int, sessionKey: bytes
    ) -> None:
        finished = Node._generateFinished(sessionKey)
        self.queueACK(addr, seqId, data=finished)

    @staticmethod
    def _generateFinished(sessionKey: bytes) -> bytes:
        return auth.generateFinished(
            sessionKey, finishedLabel=b"node finished", messages=b"\x13"
        )

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

    def queueError(
        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,
    ) -> None:
        sId = self.getSequenceId(addr)
        p = packet.ErrorPacket(
            sequence_id=sId if sId is not None else 0,
            flags=flags,
            major=major,
            minor=minor,
            data=data,
        )
        if sId is not None:
            self.incrementSequenceId(addr)
        self.queuePacket(addr, p)

    def queueDisconnect(
        self,
        addr: tuple[str, int],
        flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
        data: bytes | None = None,
    ) -> None:
        self.queueError(
            addr,
            flags=flags,
            major=error.Major.DISCONNECT,
            minor=error.DisconnectErrorCodes.DISCONNECT,
            data=data,
        )

    # receives
    def receivePacket(
        self,
    ) -> tuple[packet.Packet, tuple[str, int]] | tuple[None, None]:
        try:
            data, addr = self.socket.recvfrom(SOCKET_BUFFER_SIZE)
            try:
                p = packet.unpack(data)
                return p, addr
            except error.PacketError as e:
                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}"
                )
                major, minor = error.getErrorCod(e)
                self.queueError(addr, major, minor)
                return None, None
        except ConnectionResetError:
            return None, None
        except TimeoutError:
            return None, None

    def receive(
        self, p: packet.Packet, addr: tuple[str, int]
    ) -> tuple[packet.Packet, tuple[str, int]] | None:
        if p is not None:
            if self.handleFlags(p, addr):
                match p.packet_type:
                    case packet.Type.DEFAULT:
                        logger.info(
                            f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC} {bcolors.OKCYAN}{p}{bcolors.ENDC}"
                        )
                        return self.receiveDefault(p, addr)
                    case packet.Type.ACK:
                        logger.info(
                            f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC} {bcolors.OKCYAN}{p}{bcolors.ENDC}"
                        )
                        return self.receiveAck(p, addr)
                    case packet.Type.AUTH:
                        logger.info(
                            f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC} {bcolors.OKCYAN}{p}{bcolors.ENDC}"
                        )
                        return self.receiveAuth(p, addr)
                    case packet.Type.HEARTBEAT:
                        logger.info(
                            f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC} {bcolors.OKCYAN}{p}{bcolors.ENDC}"
                        )
                        return self.receiveHeartbeat(p, addr)
                    case packet.Type.ERROR:
                        logger.warning(
                            f"{bcolors.OKBLUE}< {addr} :{bcolors.ENDC} {bcolors.FAIL}{p}{bcolors.ENDC}"
                        )
                        try:
                            return self.receiveError(p, addr)
                        except error.PaperClipError as e:
                            self.handleError(p, addr, e)
                    case _:
                        logger.warning(
                            f"Unknown packet type '{p.packet_type}' for packet {p}"
                        )
                        self.queueError(
                            addr,
                            major=error.Major.PACKET,
                            minor=error.PacketErrorCodes.PACKET_TYPE,
```

```python
                            data=p.sequence_id,
                        )

    def receiveDefault(
        self, p: packet.Packet, addr: tuple[str, int]
    ) -> tuple[packet.Packet, tuple[str, int]]:
        self.setNewestSeqId(
            addr, self.getNewerSeqId(self.getNewestSeqId(addr), p.sequence_id)
        )
        if self.onReceiveData:
            self.onReceiveData(addr, p.data)
        return (p, addr)

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

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

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

    def receiveError(self, p: packet.ErrorPacket, addr: tuple[str, int]) ->
```

```
       None:
        raise error.getError(p.major, p.minor)(p.data)

    def listen(self) -> None:
        logger.info(
            f"{bcolors.HEADER}Listening @
               {self.socket.getsockname()}{bcolors.ENDC}"
        )
        while self.isRunning.is_set():
            p, addr = self.receivePacket()
            self.receive(p, addr)
        else:
            logger.info("| listen thread stopping...")

    # flags handle
    def handleFlags(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
        # defrag -> decrypt -> decompress -> validate checksum -> reliable
        if self.handleFrag(p, addr):
            return False
        else:
            self.handleEncrypted(p, addr)
            self.handleCompressed(p, addr)
            self.handleChecksum(p, addr)
            self.handleReliable(p, addr)
            return True

    def handleReliable(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
        if p.flags[packet.Flag.RELIABLE.value]:
            self.setNewestSeqId(
                addr, self.getNewerSeqId(self.getNewestSeqId(addr),
                   p.sequence_id)
            )
            self.setRecvAckBit(addr, p.sequence_id, True)
            self.resetRecvAckBits(addr)
            self.queueACK(addr, p.sequence_id)
            return True
        else:
            return False

    def handleFrag(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
        if p.flags[packet.Flag.FRAG.value]:
            logger.info(
                f"\t{bcolors.OKBLUE}< {addr} :{bcolors.ENDC}{bcolors.WARNING}
                   FRAG {p.fragment_id}/{p.fragment_number} {p}{bcolors.ENDC}"
            )
```

```python
            if p.sequence_id not in self.getFragBuffer(addr):
                self.getFragBuffer(addr)[p.sequence_id] = [
                    None for _ in range(p.fragment_number)
                ]
            self.getFragBuffer(addr)[p.sequence_id][p.fragment_id] = p
            if all(self.getFragBuffer(addr)[p.sequence_id]):
                defrag = p.defragment(self.getFragBuffer(addr)[p.sequence_id])
                del self.getFragBuffer(addr)[p.sequence_id]
                self.receive(defrag, addr)
            return True
        else:
            return False

    def handleCompressed(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
        if p.flags[packet.Flag.COMPRESSED.value]:
            p.decompressData()
            return True
        else:
            return False

    def handleEncrypted(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
        if p.flags[packet.Flag.ENCRYPTED.value]:
            p.decryptData(self.getSessionKey(addr))
            return True
        else:
            return False

    def handleChecksum(self, p: packet.Packet, addr: tuple[str, int]) -> bool:
        if p.flags[packet.Flag.CHECKSUM.value]:
            if not p.validateChecksum():
                logger.warning(f"\tInvalid checksum: {p}")
            else:
                logger.info(f"\tValid checksum: {p}")
            return True
        else:
            return False

    # error handle
    def handleError(
        self, p: packet.ErrorPacket, addr: tuple[str, int], e: error.PaperClipError
    ) -> None:
        match e:
            case error.ConnectionError():
                self.handleConnectionError(p, addr, e)
            case error.DisconnectError():
                self.handleDisconnectError(p, addr, e)
            case error.PacketError():
                self.handlePacketError(p, addr, e)
            case _:
                raise e

    def handleConnectionError(
        self, p: packet.ErrorPacket, addr: tuple[str, int], e:
           error.ConnectionError
    ) -> None:
        match e:
            case error.NoSpaceError():
                return self.quit("no server space", e)
            case error.CertificateInvalidError():
                return self.quit("invalid certificate", e)
            case error.FinishInvalidError():
                return self.quit("invalid finish", e)
            case _:
                raise e

    def handleDisconnectError(
        self, p: packet.ErrorPacket, addr: tuple[str, int], e:
           error.DisconnectError
    ) -> None:
        match e:
            case error.ServerDisconnectError:
                pass  # overwrite
            case error.ClientDisconnectError:
                pass  # overwrite
            case _:
                raise e

    def handlePacketError(
        self, p: packet.ErrorPacket, addr: tuple[str, int], e:
           error.PacketError
    ) -> None:
        match e:
            case error.VersionError():
                pass
            case error.PacketTypeError():
                pass
            case error.FlagsError():
```

```python
                pass
            case error.SequenceIdError():
                pass
            case error.FragmentIdError():
                pass
            case error.FragmentNumberError():
                pass
            case error.InitVectorError():
                pass
            case error.CompressionError():
                pass
            case error.ChecksumError():
                pass
            case _:
                raise e

    # util
    @staticmethod
    def packRecvAckBits(recvAckBits: list[bool], ackId: int) -> list[bool | None]:
        return [
            recvAckBits[i % 2**packet.ACK_BITS_SIZE]
            for i in range(ackId - 1, ackId - 1 - packet.ACK_BITS_SIZE, -1)
        ]

    # misc
    def startThreads(self) -> None:
        self.inboundThread.start()
        self.outboundThread.start()

    def validateCertificate(self, certificate: Certificate) -> bool:
        # overwrite
        return True

    def validateHandshake(self, finished: bytes) -> bool:
        self.handshake = Node._generateFinished(self.sessionKey) == finished
        return self.handshake

    def quit(self, msg: str = "quit call", e: Exception = None) -> None:
        logMsg = f"{bcolors.FAIL}# Quitting due to {msg}.{bcolors.ENDC}"
        if e is not None:
            logger.critical(logMsg)
        else:
            logger.info(logMsg)
        self.isRunning.clear()
        if self.inboundThread.is_alive() and get_ident() != self.inboundThread.ident:
            self.inboundThread.join()
        if self.outboundThread.is_alive() and get_ident() != self.outboundThread.ident:
            self.outboundThread.join()
        self.socket.close()
        logger.info(f"{bcolors.FAIL}# Quit finished.{bcolors.ENDC}")
        if e is not None:
            self.exitError = e
            if get_ident() in (self.inboundThread.ident, self.outboundThread.ident):
                pass
            else:
                raise e

    def _quit(self, e: Exception = None) -> None:
        self.exitError = e
        self.isRunning.clear()
```

```
# udp.packet
import struct
from enum import Enum

from cryptography.hazmat.primitives.asymmetric.ec import EllipticCurvePublicKey
from cryptography.x509 import Certificate

from . import auth, error, logger, utils

VERSION = 0
# SIZE in Bits
VERSION_SIZE = 4
PACKET_TYPE_SIZE = 4
FLAGS_SIZE = 8
SEQUENCE_ID_SIZE = 16
FRAGMENT_ID_SIZE = 8
FRAGMENT_NUM_SIZE = 8
INIT_VECTOR_SIZE = 16
CHECKSUM_SIZE = 16
ACK_ID_SIZE = SEQUENCE_ID_SIZE  # 16
ACK_BITS_SIZE = SEQUENCE_ID_SIZE  # 16

```

```

class Type(Enum):
    DEFAULT = 0
    ACK = 1
    AUTH = 2
    HEARTBEAT = 3
    ERROR = 4


class Flag(Enum):
    RELIABLE = 0
    CHECKSUM = 1
    COMPRESSED = 2
    ENCRYPTED = 3
    FRAG = 4


class Heartbeat(Enum):
    PING = 0
    PONG = 1


def lazyFlags(*fs: list[Flag]) -> list[int]:
    flags = [0 for _ in range(FLAGS_SIZE)]
    for flag in fs:
        flags[flag.value] = 1
    return flags


class Packet:
    version: int = VERSION
    packet_type: Type = Type.DEFAULT
    flags: list[int] = [0 for _ in range(FLAGS_SIZE)]
    sequence_id: int = 0
    fragment_id: int | None = None
    fragment_number: int | None = None
    init_vector: int | None = None
    checksum: int | None = None
    _data: bytes | None = None

    def __init__(
        self,
        version: int = VERSION,
        packet_type: Type = Type.DEFAULT,
        flags: list[int] = [0 for _ in range(FLAGS_SIZE)],
```

```python
        sequence_id: int = None,
        fragment_id: int | None = None,
        fragment_number: int | None = None,
        init_vector: int | None = None,
        checksum: int | None = None,
        data: bytes | None = None,
    ) -> None:
        self.version = version
        self.packet_type = packet_type
        self.flags = flags
        self.sequence_id = sequence_id
        self.fragment_id = fragment_id
        self.fragment_number = fragment_number
        self.init_vector = init_vector
        self.checksum = checksum
        self.data = data

    # util
    def encryptData(self, session_key: bytes) -> None:
        try:
            self.flags[Flag.ENCRYPTED.value] = 1
            iv = (
                self.init_vector
                if self.init_vector is not None
                else auth.generateInitVector()
            )
            cipher, iv = auth.generateCipher(session_key, iv)
            self.init_vector = iv
            self.data = auth.encryptBytes(cipher, self.data)
        except Exception as e:
            raise error.InitVectorError(e)

    def decryptData(self, session_key: bytes) -> None:
        try:
            if self.flags[Flag.ENCRYPTED.value]:
                cipher = auth.generateCipher(session_key, self.init_vector)[0]
                self.data = auth.decryptBytes(cipher, self.data)
            else:
                logger.warning(
                    f"Packet {self} is not flagged as ENCRYPTED ({self.flags})."
                )
        except Exception as e:
            raise error.InitVectorError(e)

    def compressData(self) -> None:
        try:
            self.flags[Flag.COMPRESSED.value] = 1
            self.data = utils.compressData(self.data)
        except Exception as e:
            raise error.CompressionError(e)

    def decompressData(self) -> None:
        try:
            if self.flags[Flag.COMPRESSED.value]:
                self.data = utils.decompressData(self.data)
            else:
                logger.warning(
                    f"Packet {self} is not flagged as COMPRESSED
                        ({self.flags})."
                )
        except Exception as e:
            raise error.CompressionError(e)

    def setChecksum(self) -> None:
        try:
            self.flags[Flag.CHECKSUM.value] = 1
            data = self.data if self.data is not None else b""
            self.checksum = utils.generateChecksum(data)
        except Exception as e:
            raise error.ChecksumError(e)

    def validateChecksum(self) -> bool:
        try:
            if self.flags[Flag.CHECKSUM.value]:
                data = self.data if self.data is not None else b""
                return self.checksum == utils.generateChecksum(data)
            else:
                logger.warning(
                    f"Packet {self} is not flagged as CHECKSUM
                        ({self.flags})."
                )
        except Exception as e:
            raise error.ChecksumError(e)

    @staticmethod
    def _getHeader(p) -> dict:
        header = {
            k: v
            for k, v in vars(p).items()
            if k not in ("data", "fragment_id", "fragment_number")
        }
        return header

    def fragment(self):
        self.flags[Flag.FRAG.value] = 1
        header = Packet._getHeader(self)
        fragData = utils.fragmentData(self.data)
        fragment_number = len(fragData)
        return [
            self._createFragment(
                header, fragment_id=i, fragment_number=fragment_number, data=data
            )
            for i, data in enumerate(fragData)
        ]

    @classmethod
    def _createFragment(
        cls, header: dict, fragment_id: int, fragment_number: int, data: bytes
    ):
        return cls(
            **header,
            fragment_id=fragment_id,
            fragment_number=fragment_number,
            data=data,
        )

    @classmethod
    def defragment(cls, frags):
        if frags[0].flags[Flag.FRAG.value]:
            header = Packet._getHeader(frags[0])
            header["flags"][Flag.FRAG.value] = 0
            data = utils.defragmentData([frag.data for frag in frags])
            return cls(**header, data=data)
        else:
            logger.warning(
                f"Packet {frags[0]} is not flagged as FRAG ({frags[0].flags})."
            )

    # dunder
    def __str__(self) -> str:
        try:
            s = self.pack(self)
        except error.PaperClipError:
            s = b""
        data = self.data if self.data is not None else b""
        pSize = len(s)
        dSize = len(data)
        if len(data) > 12:
            data = f"{data[:11]}...{str(data[-1:])[1:]}"
        return f"<{self.version}:{self.packet_type.name} {self.sequence_id} {''.join(map(str,self.flags))} {data} [{pSize}:{dSize}]>"

    def __eq__(self, other) -> bool:
        if isinstance(other, self.__class__):
            return vars(self) == vars(other)
        else:
            return False

    # encode / decode
    @staticmethod
    def _encodeVersion(version: int) -> int:
        try:
            return version
        except Exception as e:
            raise error.VersionError(e)

    @staticmethod
    def _decodeVersion(version: int) -> int:
        try:
            return version
        except Exception as e:
            raise error.VersionError(e)

    @staticmethod
    def _encodeType(packet_type: Type) -> int:
        try:
            return packet_type.value
        except Exception as e:
            raise error.PacketTypeError(e)

    @staticmethod
    def _decodeType(packet_type: int) -> Type:
        try:
            return Type(packet_type)
        except Exception as e:
            raise error.PacketTypeError(e)

```

```python
    @staticmethod
    def encodeVersionType(version: int, packet_type: Type) -> bytes:
        return struct.pack(
            "!B",
            (Packet._encodeVersion(version) * 16) | Packet._encodeType(packet_type),
        )

    @staticmethod
    def decodeVersionType(versionType: bytes) -> tuple[int, Type]:
        versionType = struct.unpack("!B", versionType)[0]
        version = Packet._decodeVersion(versionType >> 4)
        packet_type = Packet._decodeType(versionType & 15)
        return version, packet_type

    @staticmethod
    def encodeFlags(flags: list[int]) -> bytes:
        try:
            return struct.pack("!B", int("".join(map(str, flags)), 2))
        except Exception as e:
            raise error.FlagsError(e)

    @staticmethod
    def decodeFlags(flags: bytes) -> list[int]:
        try:
            flags = struct.unpack("!B", flags)[0]
            flags = [(flags >> i) & 1 for i in range(FLAGS_SIZE)]
            flags.reverse()
            return flags
        except Exception as e:
            raise error.FlagsError(e)

    @staticmethod
    def encodeSequenceId(sequence_id: int) -> bytes:
        try:
            return struct.pack("!I", sequence_id)
        except Exception as e:
            raise error.SequenceIdError(e)

    @staticmethod
    def decodeSequenceId(sequence_id: bytes) -> int:
        try:
            return struct.unpack("!I", sequence_id)[0]
        except Exception as e:
            raise error.SequenceIdError(e)

    @staticmethod
    def encodeFragmentId(fragment_id: int) -> bytes:
        try:
            return struct.pack("!B", fragment_id)
        except Exception as e:
            raise error.FragmentIdError(e)

    @staticmethod
    def decodeFragmentId(fragment_id: bytes) -> int:
        try:
            return struct.unpack("!B", fragment_id)[0]
        except Exception as e:
            raise error.FragmentIdError(e)

    @staticmethod
    def encodeFragmentNumber(fragment_number: int) -> bytes:
        try:
            return struct.pack("!B", fragment_number)
        except Exception as e:
            raise error.FragmentNumberError(e)

    @staticmethod
    def decodeFragmentNumber(fragment_number: bytes) -> int:
        try:
            return struct.unpack("!B", fragment_number)[0]
        except Exception as e:
            raise error.FragmentNumberError(e)

    @staticmethod
    def encodeInitVector(init_vector: bytes) -> bytes:
        try:
            return init_vector
        except Exception as e:
            raise error.InitVectorError(e)

    @staticmethod
    def decodeInitVector(init_vector: bytes) -> bytes:
        try:
            return init_vector
        except Exception as e:
            raise error.InitVectorError(e)

    @staticmethod
    def encodeChecksum(checksum: int) -> bytes:
        try:
            return struct.pack("!I", checksum)
        except Exception as e:
            raise error.ChecksumError(e)

    @staticmethod
    def decodeChecksum(checksum: bytes) -> int:
        try:
            return struct.unpack("!I", checksum)[0]
        except Exception as e:
            raise error.ChecksumError(e)

    @staticmethod
    def encodeHeader(
        version: int,
        packet_type: Type,
        flags: list[int],
        sequence_id: int,
        fragment_id: int | None = None,
        fragment_number: int | None = None,
        init_vector: int | None = None,
        checksum: int | None = None,
    ) -> bytes:
        versionType = Packet.encodeVersionType(version, packet_type)
        flags = Packet.encodeFlags(flags)
        sequence_id = Packet.encodeSequenceId(sequence_id)
        fragment_id = (
            Packet.encodeFragmentId(fragment_id) if fragment_id is not None else b""
        )
        fragment_number = (
            Packet.encodeFragmentNumber(fragment_number)
            if fragment_number is not None
            else b""
        )
        init_vector = (
            Packet.encodeInitVector(init_vector) if init_vector is not None else b""
        )
        checksum = Packet.encodeChecksum(checksum) if checksum is not None else b""
        return (
            versionType
            + flags
            + sequence_id
            + fragment_id
            + fragment_number
            + init_vector
            + checksum
        )

    @staticmethod
    def decodeHeader(
        header: bytes,
    ) -> tuple[
        int, Type, list[int], int, int | None, int | None, int | None, int | None, int
    ]:
        version, packet_type = Packet.decodeVersionType(header[0:1])
        flags = Packet.decodeFlags(header[1:2])
        sequence_id = Packet.decodeSequenceId(header[2:6])
        offset = 6
        if flags[Flag.FRAG.value]:
            fragment_id = Packet.decodeFragmentId(header[offset : offset + 1])
            fragment_number = Packet.decodeFragmentNumber(
                header[offset + 1 : offset + 2]
            )
            offset += 2
        else:
            fragment_id = None
            fragment_number = None
        if flags[Flag.ENCRYPTED.value]:
            init_vector = Packet.decodeInitVector(header[offset : offset + 16])
            offset += 16
        else:
            init_vector = None
        if flags[Flag.CHECKSUM.value]:
            checksum = Packet.decodeChecksum(header[offset : offset + 4])
            offset += 4
        else:
            checksum = None
        return (
            version,
            packet_type,
            flags,
            sequence_id,
            fragment_id,
            fragment_number,
            init_vector,
            checksum,
            offset,
        )

    # pack / unpack
    @classmethod
    def _packHeader(cls, p) -> bytes:
        header = cls.encodeHeader(
            p.version,
            p.packet_type,
            p.flags,
            p.sequence_id,
            p.fragment_id,
            p.fragment_number,
            p.init_vector,
            p.checksum,
        )
        return header

    @classmethod
    def pack(cls, p) -> bytes:
        header = cls._packHeader(p)
        data = p.data if p.data is not None else b""
        return header + data

    @classmethod
    def _unpackHeader(cls, bytesP: bytes):
        *header, offset = cls.decodeHeader(bytesP)
        return *header, offset

    @classmethod
    def unpack(cls, bytesP: bytes):
        *header, offset = cls._unpackHeader(bytesP)
        data = bytesP[offset:] if offset < len(bytesP) else None
        return cls(*header, data=data)


class AckPacket(Packet):
    ack_id: int = 0
    ack_bits: list[int | None] = [None for _ in range(ACK_BITS_SIZE)]

    def __init__(
        self,
        version: int = VERSION,
        packet_type: Type.ACK = Type.ACK,
        flags: list[int] = [0 for _ in range(FLAGS_SIZE)],
        sequence_id: int = None,
        fragment_id: int | None = None,
        fragment_number: int | None = None,
        init_vector: int | None = None,
        checksum: int | None = None,
        ack_id: int = None,
        ack_bits: list[int | None] = [None for _ in range(ACK_BITS_SIZE)],
        data: bytes | None = None,
    ) -> None:
        super().__init__(
            version,
            Type.ACK,
            flags,
            sequence_id,
            fragment_id,
            fragment_number,
            init_vector,
            checksum,
            data,
        )
        self.ack_id = ack_id
        self.ack_bits = ack_bits

    # dunder
    def __str__(self) -> str:
        s = self.pack(self)
        data = self.data if self.data is not None else b""
        pSize = len(s)
        dSize = len(data)
        if len(data) > 12:
            data = f"{data[:11]}...{str(data[-1:])[1:]}"
        return f"<{self.version}:{self.packet_type.name}
           {self.sequence_id}:{self.ack_id} {''.join(map(str,self.flags))}
           {data} [{pSize}:{dSize}]>"

    # encode / decode
    @staticmethod
    def encodeAckId(ack_id: int) -> bytes:
        return struct.pack("!I", ack_id)

    @staticmethod
    def decodeAckId(ack_id: bytes) -> int:
        return struct.unpack("!I", ack_id)[0]

    @staticmethod
    def encodeAckBits(ack_bits: list[int]) -> bytes:
        return struct.pack(
            "!I",
            int(
                "".join(
                    map(str, (int(bit) if bit is not None else 0 for bit in ack_bits))
                ),
                2,
            ),
        )

    @staticmethod
    def decodeAckBits(ack_bits: bytes) -> list[int]:
        ack_bits = struct.unpack("!I", ack_bits)[0]
        ack_bits = [(ack_bits >> i) & 1 for i in range(ACK_BITS_SIZE)]
        ack_bits.reverse()
        return ack_bits

    @staticmethod
    def encodeHeader(
        version: int,
        packet_type: Type,
        flags: list[int],
        sequence_id: int,
        fragment_id: int | None = None,
        fragment_number: int | None = None,
        init_vector: int | None = None,
        checksum: int | None = None,
        ack_id: int = 0,
        ack_bits: list[int | None] = [None for _ in range(ACK_BITS_SIZE)],
    ) -> bytes:
        header = Packet.encodeHeader(
            version,
            packet_type,
            flags,
            sequence_id,
            fragment_id,
            fragment_number,
            init_vector,
            checksum,
        )
        ack_id = AckPacket.encodeAckId(ack_id)
        ack_bits = AckPacket.encodeAckBits(ack_bits)
```

```python
        return header + ack_id + ack_bits

    @staticmethod
    def decodeHeader(
        header: bytes,
    ) -> tuple[
        int,
        Type,
        list[int],
        int,
        int | None,
        int | None,
        int | None,
        int | None,
        int,
        list[int | None],
        int,
    ]:
        *h, offset = Packet.decodeHeader(header)
        ack_id = AckPacket.decodeAckId(header[offset : offset + 4])
        offset += 4
        ack_bits = AckPacket.decodeAckBits(header[offset : offset + 4])
        offset += 4
        return *h, ack_id, ack_bits, offset

    # pack / unpack
    @classmethod
    def _packHeader(cls, p) -> bytes:
        header = cls.encodeHeader(
            p.version,
            p.packet_type,
            p.flags,
            p.sequence_id,
            p.fragment_id,
            p.fragment_number,
            p.init_vector,
            p.checksum,
            p.ack_id,
            p.ack_bits,
        )
        return header


class AuthPacket(Packet):
    _public_key_size: int | None = None
    public_key: EllipticCurvePublicKey | None = None
    _certificate_size: int | None = None
    certificate: Certificate | None = None

    def __init__(
        self,
        version: int = VERSION,
        packet_type: Type = Type.AUTH,
        flags: list[int] = [0 for _ in range(FLAGS_SIZE)],
        sequence_id: int = None,
        fragment_id: int | None = None,
        fragment_number: int | None = None,
        init_vector: int | None = None,
        checksum: int | None = None,
        public_key_size: int | None = None,
        public_key: EllipticCurvePublicKey = None,
        certificate_size: int | None = None,
        certificate: Certificate | None = None,
    ) -> None:
        super().__init__(
            version,
            Type.AUTH,
            flags,
            sequence_id,
            fragment_id,
            fragment_number,
            init_vector,
            checksum,
            data=None,
        )
        self.public_key_size = public_key_size
        self.public_key = public_key
        self.certificate_size = certificate_size
        self.certificate = certificate

    # setter / getter
    @property
    def public_key_size(self) -> int | None:
        if self._public_key_size is None:
            self.public_key_size = (
                AuthPacket.getPublicKeyBytesSize(self.public_key)
                if self.public_key is not None
                else None
            )
        return self._public_key_size
```

```python

    @public_key_size.setter
    def public_key_size(self, v: int | None) -> None:
        self._public_key_size = v

    @staticmethod
    def getPublicKeyBytesSize(publicKey: EllipticCurvePublicKey) -> int:
        return len(auth.getDerFromPublicEc(publicKey))

    @property
    def certificate_size(self) -> int | None:
        if self._certificate_size is None:
            self.certificate_size = (
                self.getCertificateByteSize(self.certificate)
                if self.certificate is not None
                else None
            )
        return self._certificate_size

    @certificate_size.setter
    def certificate_size(self, v: int | None) -> None:
        self._certificate_size = v

    @staticmethod
    def getCertificateByteSize(certificate: Certificate) -> int:
        return len(auth.getDerFromCertificate(certificate))

    # encode / decode
    @staticmethod
    def encodePublicKeySize(public_key_size: int) -> bytes:
        return struct.pack("!B", public_key_size)

    @staticmethod
    def decodePublicKeySize(public_key_size: bytes) -> int:
        return struct.unpack("!B", public_key_size)[0]

    @staticmethod
    def encodePublicKey(public_key: EllipticCurvePublicKey) -> bytes:
        return auth.getDerFromPublicEc(public_key)

    @staticmethod
    def decodePublicKey(public_key: bytes) -> EllipticCurvePublicKey:
        return auth.getPublicEcFromDer(public_key)

    @staticmethod
    def encodeCertificateSize(certificate_size: int) -> bytes:
        return struct.pack("!H", certificate_size)

    @staticmethod
    def decodeCertificateSize(certificate_size: bytes) -> int:
        return struct.unpack("!H", certificate_size)[0]

    @staticmethod
    def encodeCertificate(certificate: Certificate) -> bytes:
        return auth.getDerFromCertificate(certificate)

    @staticmethod
    def decodeCertificate(certificate: bytes) -> Certificate:
        return auth.getCertificateFromDer(certificate)

    @staticmethod
    def encodeHeader(
        version: int,
        packet_type: Type,
        flags: list[int],
        sequence_id: int,
        fragment_id: int | None = None,
        fragment_number: int | None = None,
        init_vector: int | None = None,
        checksum: int | None = None,
        public_key_size: int | None = None,
        public_key: EllipticCurvePublicKey | None = None,
        certificate_size: int | None = None,
        certificate: Certificate | None = None,
    ) -> bytes:
        header = Packet.encodeHeader(
            version,
            packet_type,
            flags,
            sequence_id,
            fragment_id,
            fragment_number,
            init_vector,
            checksum,
        )
        public_key_size = AuthPacket.encodePublicKeySize(public_key_size)
        public_key = AuthPacket.encodePublicKey(public_key)
        certificate_size = (
            AuthPacket.encodeCertificateSize(certificate_size)
            if certificate_size is not None
```

```python
            else b""
        )
        certificate = (
            AuthPacket.encodeCertificate(certificate)
            if certificate is not None
            else b""
        )
        return header + public_key_size + public_key + certificate_size + certificate

    @staticmethod
    def decodeHeader(
        header: bytes,
    ) -> tuple[
        int,
        Type,
        list[int],
        int,
        int | None,
        int | None,
        int | None,
        int | None,
        int,
        EllipticCurvePublicKey,
        int | None,
        Certificate | None,
        int,
    ]:
        *h, offset = Packet.decodeHeader(header)
        public_key_size = AuthPacket.decodePublicKeySize(header[offset : offset + 1])
        offset += 1
        public_key = AuthPacket.decodePublicKey(
            header[offset : offset + public_key_size]
        )
        offset += public_key_size
        if offset < len(header):  # check if more bytes left to decode
            certificate_size = AuthPacket.decodeCertificateSize(
                header[offset : offset + 2]
            )
            offset += 2
            certificate = AuthPacket.decodeCertificate(
                header[offset : offset + certificate_size]
            )
            offset += certificate_size
        else:
            certificate_size = None
            certificate = None
        return *h, public_key_size, public_key, certificate_size, certificate, offset

    # pack / unpack
    @classmethod
    def _packHeader(cls, p) -> bytes:
        header = cls.encodeHeader(
            p.version,
            p.packet_type,
            p.flags,
            p.sequence_id,
            p.fragment_id,
            p.fragment_number,
            p.init_vector,
            p.checksum,
            p.public_key_size,
            p.public_key,
            p.certificate_size,
            p.certificate,
        )
        return header

    @classmethod
    def unpack(cls, bytesP: bytes):
        *header, offset = cls._unpackHeader(bytesP)
        return cls(*header)


class HeartbeatPacket(Packet):
    heartbeat: bool

    def __init__(
        self,
        version: int = VERSION,
        packet_type: Type = Type.HEARTBEAT,
        flags: list[int] = [0 for _ in range(FLAGS_SIZE)],
        sequence_id: int = None,
        fragment_id: int | None = None,
        fragment_number: int | None = None,
        init_vector: int | None = None,
        checksum: int | None = None,
        heartbeat: bool = 0,
```

```python
        data: bytes | None = None,
    ) -> None:
        super().__init__(
            version,
            Type.HEARTBEAT,
            flags,
            sequence_id,
            fragment_id,
            fragment_number,
            init_vector,
            checksum,
            data,
        )
        self.heartbeat = heartbeat

    # encode / decode
    @staticmethod
    def encodeHeartbeat(heartbeat: bool) -> bytes:
        return struct.pack("!?", heartbeat)

    @staticmethod
    def decodeHeartbeat(heartbeat: bytes) -> bool:
        return struct.unpack("!?", heartbeat)[0]

    @staticmethod
    def encodeHeader(
        version: int,
        packet_type: Type,
        flags: list[int],
        sequence_id: int,
        fragment_id: int | None = None,
        fragment_number: int | None = None,
        init_vector: int | None = None,
        checksum: int | None = None,
        heartbeat: bool = 0,
    ) -> bytes:
        header = Packet.encodeHeader(
            version,
            packet_type,
            flags,
            sequence_id,
            fragment_id,
            fragment_number,
            init_vector,
            checksum,
        )
        heartbeat = HeartbeatPacket.encodeHeartbeat(heartbeat)
        return header + heartbeat

    @staticmethod
    def decodeHeader(
        header: bytes,
    ) -> tuple[
        int,
        Type,
        list[int],
        int,
        int | None,
        int | None,
        int | None,
        int | None,
        bool,
        int,
    ]:
        *h, offset = Packet.decodeHeader(header)
        heartbeat = HeartbeatPacket.decodeHeartbeat(header[offset : offset + 1])
        offset += 1
        return *h, heartbeat, offset

    # pack / unpack
    @classmethod
    def _packHeader(cls, p) -> bytes:
        header = cls.encodeHeader(
            p.version,
            p.packet_type,
            p.flags,
            p.sequence_id,
            p.fragment_id,
            p.fragment_number,
            p.init_vector,
            p.checksum,
            p.heartbeat,
        )
        return header


class ErrorPacket(Packet):
    _major: error.Major
    _minor: error.Minor

    def __init__(
        self,
        version: int = VERSION,
        packet_type: Type = Type.ERROR,
        flags: list[int] = [0 for _ in range(FLAGS_SIZE)],
        sequence_id: int = None,
        fragment_id: int | None = None,
        fragment_number: int | None = None,
        init_vector: int | None = None,
        checksum: int | None = None,
        major: error.Major | int = error.Major.ERROR,
        minor: error.Minor | int = 0,
        data: bytes | None = None,
    ) -> None:
        super().__init__(
            version,
            Type.ERROR,
            flags,
            sequence_id,
            fragment_id,
            fragment_number,
            init_vector,
            checksum,
            data,
        )
        self.major = major
        self.minor = minor

    @property
    def major(self) -> error.Major:
        return self._major

    @major.setter
    def major(self, v: error.Major | int):
        if isinstance(v, error.Major):
            self._major = v
        else:
            self._major = error.Major(v)

    @property
    def minor(self) -> error.Minor:
        return self._minor

    @minor.setter
```

```python
    def minor(self, v: error.Minor | int):
        if isinstance(v, error.Minor):
            self._minor = v
        else:
            self._minor = error.getMinor(self.major, v)

    # dunder
    def __str__(self) -> str:
        s = self.pack(self)
        data = self.data if self.data is not None else b""
        pSize = len(s)
        dSize = len(data)
        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}]>"

    # encode / decode
    @staticmethod
    def _encodeMajor(major: error.Major) -> int:
        return major.value

    @staticmethod
    def _decodeMajor(major: int) -> error.Major:
        return error.Major(major)

    @staticmethod
    def _encodeMinor(minor: error.Minor) -> int:
        return minor.value if minor != error.Minor else 0

    @staticmethod
    def _decodeMinor(major: error.Major, minor: int) -> error.Minor:
        return error.getMinor(major, minor)

    def encodeMajorMinor(major: int, minor: int) -> bytes:
        majorMinor = (ErrorPacket._encodeMajor(major) * 16) | ErrorPacket._encodeMinor(
            minor
        )
        return struct.pack("!B", majorMinor)

    def decodeMajorMinor(majorMinor: bytes) -> tuple[int, int]:
        majorMinor = struct.unpack("!B", majorMinor)[0]
        major = ErrorPacket._decodeMajor(majorMinor >> 4)
        minor = ErrorPacket._decodeMinor(major, majorMinor & 15)
        return major, minor

    @staticmethod
    def encodeHeader(
        version: int,
        packet_type: Type,
        flags: list[int],
        sequence_id: int,
        fragment_id: int | None = None,
        fragment_number: int | None = None,
        init_vector: int | None = None,
        checksum: int | None = None,
        major: int = 0,
        minor: int = 0,
    ) -> bytes:
        header = Packet.encodeHeader(
            version,
            packet_type,
            flags,
            sequence_id,
            fragment_id,
            fragment_number,
            init_vector,
            checksum,
        )
        majorMinor = ErrorPacket.encodeMajorMinor(major, minor)
        return header + majorMinor

    @staticmethod
    def decodeHeader(
        header: bytes,
    ) -> tuple[
        int,
        Type,
        list[int],
        int,
        int | None,
        int | None,
        int | None,
        int | None,
        int,
        int,
        int,
    ]:
        *h, offset = Packet.decodeHeader(header)
        major, minor = ErrorPacket.decodeMajorMinor(header[offset : offset +
           1])
        offset += 1
        return *h, major, minor, offset

    # pack / unpack
    @classmethod
    def _packHeader(cls, p) -> bytes:
        header = cls.encodeHeader(
            p.version,
            p.packet_type,
            p.flags,
            p.sequence_id,
            p.fragment_id,
            p.fragment_number,
            p.init_vector,
            p.checksum,
            p.major,
            p.minor,
        )
        return header


def unpack(rawP:bytes) -> Packet:
    packet_type = Packet.decodeVersionType(rawP[0:1])[1]
    match packet_type:
        case Type.DEFAULT:
            return Packet.unpack(rawP)
        case Type.ACK:
            return AckPacket.unpack(rawP)
        case Type.AUTH:
            return AuthPacket.unpack(rawP)
        case Type.HEARTBEAT:
            return HeartbeatPacket.unpack(rawP)
        case Type.ERROR:
            return ErrorPacket.unpack(rawP)
        case _:
            logger.warning(f"Cannot unpack '{packet_type}' due to invalid packet type.")
```

```
# udp.server
import base64
import json
import socket
```

```python
import time
from datetime import datetime
from threading import Event, Lock, Thread

import requests
from cryptography.hazmat.primitives.asymmetric.rsa import RSAPrivateKey

from . import (
    HEARTBEAT_MAX_TIME,
    HEARTBEAT_MIN_TIME,
    MAX_CLIENTS,
    auth,
    bcolors,
    error,
    logger,
    node,
    packet,
)


class Server(node.Node):
    clients: dict[tuple[str, int], node.Node]
    clientsLock: Lock
    clientDeleteEvent: Event
    rsaKey: RSAPrivateKey | None
    heartbeatThread: Thread
    onClientJoin: None
    onClientLeave: None
    maxClients: int

    def __init__(
        self,
        addr,
        maxClients: int = MAX_CLIENTS,
        rsaKey: RSAPrivateKey | None = None,
        onClientJoin=None,
        onClientLeave=None,
        onReceiveData=None,
    ) -> None:
        self.clients = {}
        self.clientsLock = Lock()
        self.clientDeleteEvent = Event()
        self.clientDeleteEvent.set()
        self.rsaKey = rsaKey if rsaKey is not None else auth.generateRsaKey()
        self.onClientJoin = onClientJoin
        self.onClientLeave = onClientLeave
        self.maxClients = maxClients
        s = socket.socket(type=socket.SOCK_DGRAM)
        super().__init__(
            addr,
            cert=auth.generateUserCertificate(self.rsaKey),
            socket=s,
            onReceiveData=onReceiveData,
        )
        self.heartbeatThread = Thread(
            name=f"{self.port}:Heartbeat", target=self.heartbeat, daemon=True
        )
        self.bind(self.addr)

    def receiveAck(self, p: packet.AckPacket, addr: tuple[str, int]) -> None:
        super().receiveAck(p, addr)
        if p.data is not None and not self.getHandshake(
            addr
        ):  # ack has payload & client has not completed handshake => validate handshake
            if not self.validateHandshake(addr, p.data):
                # raise ValueError(f"Local finished value does not match peer finished value {p.data}")
                logger.error(
                    f"Local finished value does not match peer finished value {p.data}"
                )
                self.queueError(
                    addr,
                    major=error.Major.CONNECTION,
                    minor=error.ConnectionErrorCodes.FINISH_INVALID,
                    data=b"Invalid finish.",
                )
            else:
                # print(f"{bcolors.OKGREEN}# Handshake with {addr} successful.{bcolors.ENDC}")
                logger.info(
                    f"{bcolors.OKGREEN}# Handshake with {addr} successful.{bcolors.ENDC}"
                )
                if self.onClientJoin:
                    self.onClientJoin(addr, self.getClientId(addr))

    def receiveAuth(
        self, p: packet.AuthPacket, addr: tuple[str, int]
    ) -> tuple[packet.AuthPacket, tuple[str, int]]:
        if addr not in self.clients:  # new client
            if self.isNotFull():  # check space
                # print(f"{bcolors.WARNING}# Handshake with {addr} starting.{bcolors.ENDC}")
                logger.info(
                    f"{bcolors.WARNING}# Handshake with {addr} starting.{bcolors.ENDC}"
                )
                valid, accountId = self.validateCertificate(p.certificate)
                if not valid:
                    # raise ValueError(f"Invalid peer cert {p.certificate}")
                    logger.error(f"Invalid peer cert {p.certificate}")
                    self.queueError(
                        addr,
                        major=error.Major.CONNECTION,
                        minor=error.ConnectionErrorCodes.CERTIFICATE_INVALID,
                        data=b"Invalid Certificate.",
                    )
                else:
                    self.makeClient(addr, p.certificate, accountId)
                    self.regenerateEcKey(addr)
                    sessionKey = auth.generateSessionKey(
                        self.getEcKey(addr), p.public_key
                    )
                    self.setSessionKey(addr, sessionKey)
                    self.queueAuth(addr, self.cert, self.getEcKey(addr).public_key())
                    self.queueFinished(addr, p.sequence_id, self.getSessionKey(addr))
            else:
                # print(f"{bcolors.FAIL}# Handshake with {addr} denied due to NO_SPACE.{bcolors.ENDC}")
                logger.warning(
                    f"{bcolors.FAIL}# Handshake with {addr} denied due to NO_SPACE.{bcolors.ENDC}"
                )
                self.queueError(
                    addr,
                    major=error.Major.CONNECTION,
                    minor=error.ConnectionErrorCodes.NO_SPACE,
                    data=b"Server is Full.",
                )
        else:
            sessionKey = auth.generateSessionKey(self.getEcKey(addr),
               p.public_key)
        if addr in self.clients:
            if self.getSessionKey(addr) != sessionKey:  # new client sessionKey
                # print(f"{bcolors.WARNING}# Handshake with {addr} reset.{bcolors.ENDC}")
                logger.info(
                    f"{bcolors.WARNING}# Handshake with {addr} reset.{bcolors.ENDC}"
                )
                valid, accountId = self.validateCertificate(p.certificate)
                if not valid:
                    # raise ValueError(f"Invalid peer cert {p.certificate}")
                    logger.warning(f"Invalid peer cert {p.certificate}")
                    self.queueError(
                        addr,
                        major=error.Major.CONNECTION,
                        minor=error.ConnectionErrorCodes.CERTIFICATE_INVALID,
                        data=b"Invalid Certificate.",
                    )
                else:
                    self.regenerateEcKey(addr)
                    # self.clients[addr].cert = p.certificate # shouldn't change
                    sessionKey = auth.generateSessionKey(
                        self.getEcKey(addr), p.public_key
                    )
                    self.setSessionKey(addr, sessionKey)  # make new session key
                    self.queueAuth(addr, self.cert, self.getEcKey(addr).public_key())
                    self.queueFinished(addr, p.sequence_id, self.getSessionKey(addr))
        return (p, addr)

    def queueDisconnect(
        self,
        flags: list[bool] = [0 for _ in range(packet.FLAGS_SIZE)],
        data: bytes | None = None,
    ):
        with self.clientsLock:
            clientAddrs = [addr for addr in self.clients]
        for addr in clientAddrs:
            self.queueError(
                addr,
```

```python
                flags=flags,
                major=error.Major.DISCONNECT,
                minor=error.DisconnectErrorCodes.SERVER_DISCONNECT,
                data=data,
            )

    def getSessionKey(self, clientAddr: tuple[str, int]) -> bytes | None:
        with self.clientsLock:
            return self.clients[clientAddr].sessionKey

    def setSessionKey(self, clientAddr: tuple[str, int], sessionKey: bytes) -> None:
        with self.clientsLock:
            self.clients[clientAddr].sessionKey = sessionKey

    def getHandshake(self, clientAddr: tuple[str, int]) -> bool:
        with self.clientsLock:
            return self.clients[clientAddr].handshake

    def getSentAckBit(self, clientAddr: tuple[str, int], p: packet.Packet) -> bool:
        with self.clientsLock:
            return self.clients[clientAddr].sentAckBits[p.sequence_id]

    def setSentAckBit(self, clientAddr: tuple[str, int], ackBit: int, v: bool) -> None:
        with self.clientsLock:
            self.clients[clientAddr].sentAckBits[ackBit] = v

    def getSentAckBits(self, clientAddr: tuple[str, int]) -> list[bool]:
        with self.clientsLock:
            return self.clients[clientAddr].sentAckBits

    def getRecvAckBit(self, clientAddr: tuple[str, int], p: packet.Packet) -> bool:
        with self.clientsLock:
            return self.clients[clientAddr].recvAckBits[p.sequence_id]

    def getRecvAckBits(self, clientAddr: tuple[str, int]) -> list[bool]:
        with self.clientsLock:
            return self.clients[clientAddr].recvAckBits

    def setRecvAckBit(self, clientAddr: tuple[str, int], ackBit: int, v: bool) -> None:
        with self.clientsLock:
            self.clients[clientAddr].recvAckBits[ackBit] = v

    def getNewestSeqId(self, clientAddr: tuple[str, int]) -> int:
        with self.clientsLock:
            if clientAddr in self.clients:
                return self.clients[clientAddr].newestSeqId
            else:
                return 0

    def setNewestSeqId(self, clientAddr: tuple[str, int], newSeqId: int) -> None:
        with self.clientsLock:
            if clientAddr in self.clients:
                self.clients[clientAddr].newestSeqId = newSeqId

    def getFragBuffer(
        self, clientAddr: tuple[str, int]
    ) -> dict[int, list[packet.Packet]]:
        with self.clientsLock:
            return self.clients[clientAddr].fragBuffer

    def getEcKey(self, clientAddr: tuple[str, int]) -> auth.ec.EllipticCurvePrivateKey:
        with self.clientsLock:
            return self.clients[clientAddr].ecKey

    def getSequenceId(self, clientAddr: tuple[str, int]) -> int | None:
        with self.clientsLock:
            return (
                self.clients[clientAddr].sequenceId
                if clientAddr in self.clients
                else None
            )

    def getQueue(self, clientAddr: tuple[str, int]) -> node.Queue:
        with self.clientsLock:
            return (
                self.clients[clientAddr].queue
                if clientAddr in self.clients
                else self.queue
            )

    def getSequenceIdLock(self, clientAddr: tuple[str, int]) -> Lock:
        with self.clientsLock:
            return self.clients[clientAddr].sequenceIdLock

    def incrementSequenceId(self, clientAddr: tuple[str, int]) -> None:
        with self.getSequenceIdLock(clientAddr):
            with self.clientsLock:
                self.clients[clientAddr].sequenceId += 1

    def getHeartbeat(self, clientAddr: tuple[str, int]) -> datetime:
        with self.clientsLock:
            return self.clients[clientAddr].heartbeat

    def setHeartbeat(self, clientAddr: tuple[str, int], v: datetime) -> None:
        with self.clientsLock:
            self.clients[clientAddr].heartbeat = v

    def regenerateEcKey(self, clientAddr: tuple[str, int]) -> None:
        with self.clientsLock:
            self.clients[clientAddr].regenerateEcKey()

    def checkClientExists(self, clientAddr: tuple[str, int]) -> bool:
        with self.clientsLock:
            return clientAddr in self.clients

    def validateHandshake(self, clientAddr: tuple[str, int], finished: bytes) -> bool:
        with self.clientsLock:
            return self.clients[clientAddr].validateHandshake(finished)

    def getClientLength(self) -> int:
        with self.clientsLock:
            return len(self.clients)

    def getClientId(self, clientAddr: tuple[str, int]) -> int:
        with self.clientsLock:
            return self.clients[clientAddr].accountId

    def getClientIds(self) -> list[int]:
        with self.clientsLock:
            return [client.id for addr, client in self.clients.items()]

    def isNotFull(self) -> bool:
        with self.clientsLock:
            return len(self.clients) < self.maxClients  # check space

    def isEmpty(self) -> bool:
        with self.clientsLock:
            return len(self.clients) == 0

    def listen(self) -> None:
        logger.info(
            f"{bcolors.HEADER}Listening @ {self.socket.getsockname()}{bcolors.ENDC}"
        )
        while self.isRunning.is_set():
            p, addr = self.receivePacket()
            if p is not None and addr is not None:
                if self.checkClientExists(addr):  # client exists
                    self.setHeartbeat(addr, datetime.now())
                    if self.getHandshake(
                        addr
                    ):  # client handshake complete => allow all packet types
                        self.receive(p, addr)
                    else:
                        if (
                            p.packet_type
                            in (packet.Type.AUTH, packet.Type.ACK, packet.Type.ERROR)
                        ):  # client handshake incomplete => drop all non-AUTH | non-ACK | non-ERROR packets
                            self.receive(p, addr)
                else:
                    if p.packet_type in (
                        packet.Type.AUTH,
                        packet.Type.ERROR,
                    ):  # client not exists => drop all non-AUTH | non-ERROR packets
                        self.receive(p, addr)
                    else:
                        logger.warning(
                            f"{bcolors.WARNING}! {addr} :{bcolors.ENDC} {bcolors.WARNING}{p}{bcolors.ENDC}"
                        )
        else:
            logger.info("| listen thread stopping...")

    def heartbeat(self) -> None:
        while self.isRunning.is_set():
            time.sleep(HEARTBEAT_MIN_TIME)
            with self.clientsLock:
                clients = [k for k in self.clients.keys()]
            for clientAddr in clients:
                heartbeat = self.getHeartbeat(clientAddr)
                delta = (datetime.now() - heartbeat).seconds
                if delta > HEARTBEAT_MAX_TIME:
                    self.removeClient(
                        clientAddr,
                        debugStr=f"due to heartbeat timeout (last contact was {heartbeat})",
                    )
                elif delta > HEARTBEAT_MIN_TIME:
                    self.queueHeartbeat(clientAddr, heartbeat=False)
        else:
            logger.info("| heartbeat thread stopping...")

    def makeClient(
        self, clientAddr: tuple[str, int], cert: auth.x509.Certificate, accountId: int
    ) -> None:
        c = node.Node(
            clientAddr,
            cert=cert,
            accountId=accountId,
            sendLock=self.sendLock,
            socket=self.socket,
        )
        c.outboundThread.start()
        with self.clientsLock:
            self.clients[clientAddr] = c

    def removeClient(self, clientAddr: tuple[str, int], debugStr="") -> None:
        if self.checkClientExists(clientAddr):
            cId = self.getClientId(clientAddr)
            with self.clientsLock:
                logger.info(
                    f"{bcolors.FAIL}# Client {clientAddr} was removed{' '+debugStr}.{bcolors.ENDC}"
                )
                self.clients[clientAddr].isRunning.clear()
                del self.clients[clientAddr]
                if self.onClientLeave:
                    self.onClientLeave(clientAddr, cId)

    # misc
    def startThreads(self) -> None:
        super().startThreads()
        self.heartbeatThread.start()

    def validateCertificate(self, certificate: auth.x509.Certificate) -> bool:
        url = f"http://{self.host}:5000/auth/certificate/validate"
        headers = {"Content-Type": "application/json"}
        certificate = base64.encodebytes(
            auth.getDerFromCertificate(certificate)
        ).decode()
        data = {"certificate": certificate}
        try:
            r = requests.get(url, headers=headers, data=json.dumps(data))
            if r.status_code == 200:
                return r.json()["valid"], r.json()["account-id"]
            else:
                return False
        except:  # noqa: E722
            # Cert server unresponsive
            return False

    def quit(
        self, msg: str = "quit call", e: Exception | None = None
    ) -> Exception | None:
        self.queueDisconnect(data=msg.encode())
        self.queue.join()
        e = super().quit(msg, e)
        if self.heartbeatThread.is_alive:
            self.heartbeatThread.join()
        return e

    def handleDisconnectError(
        self, p: packet.ErrorPacket, addr: tuple[str, int], e: error.DisconnectError
    ) -> None:
        match e:
            case error.ServerDisconnectError():
                pass  # should not react to server disconnect
            case error.ClientDisconnectError():
                self.removeClient(addr, "The client has closed")
            case _:
                raise e
```

```
# udp.utils
import zlib

```

```python
from . import MAX_FRAGMENT_SIZE


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


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


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


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


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

9.3.2 server

```python
# server.__init__
import os

import dotenv
from flask import Flask

from udp import logger  # noqa: F401

from .models import *  # noqa: F403

from sqlalchemy_utils import database_exists, create_database

dotenv.load_dotenv()
```

```
PRUNE_TIME = int(os.environ.get("PRUNE_TIME"))


def create_app():
    app = Flask(__name__)

    app.jinja_env.trim_blocks = True
    app.jinja_env.lstrip_blocks = True

    app.config["SECRET_KEY"] = os.environ.get("SECRET_KEY").encode()
    uri = os.environ.get("SQLALCHEMY_DATABASE_URI")
    _init = False
    if not database_exists(uri):
        _init = True
        create_database(uri)
    app.config["SQLALCHEMY_DATABASE_URI"] = uri

    db.init_app(app)  # noqa: F405

    with app.app_context():
        db.create_all()  # noqa: F405

    if _init:
        with app.app_context():
            # init games
            from rps import ID, NAME, MIN_PLAYERS, MAX_PLAYERS
            Statement.createGame(ID, NAME, MIN_PLAYERS, MAX_PLAYERS)  # noqa: F405
            # example accounts
            m = Statement.createAccount("Mario", "ItsAMe123")  # noqa: F405
            p = Statement.createAccount("Peach", "MammaMia!")  # noqa: F405
            b = Statement.createAccount("Bowser", "M4r10SucK5")  # noqa: F405
            Statement.createFriends(m.id, p.id)  # noqa: F405
            Statement.createFriends(p.id, b.id)  # noqa: F405

    from .main import main as main_blueprint

    app.register_blueprint(main_blueprint)

    return app
```

```
# server.lobbies
import os
```

```python

import dotenv
from flask import Flask

from udp import logger  # noqa: F401

from .models import *  # noqa: F403

from sqlalchemy_utils import database_exists, create_database

dotenv.load_dotenv()
PRUNE_TIME = int(os.environ.get("PRUNE_TIME"))


def create_app():
    app = Flask(__name__)

    app.jinja_env.trim_blocks = True
    app.jinja_env.lstrip_blocks = True

    app.config["SECRET_KEY"] = os.environ.get("SECRET_KEY").encode()
    uri = os.environ.get("SQLALCHEMY_DATABASE_URI")
    _init = False
    if not database_exists(uri):
        _init = True
        create_database(uri)
    app.config["SQLALCHEMY_DATABASE_URI"] = uri

    db.init_app(app)  # noqa: F405

    with app.app_context():
        db.create_all()  # noqa: F405

    if _init:
        with app.app_context():
            # init games
            from rps import ID, NAME, MIN_PLAYERS, MAX_PLAYERS
            Statement.createGame(ID, NAME, MIN_PLAYERS, MAX_PLAYERS)  # noqa: F405
            # example accounts
            m = Statement.createAccount("Mario", "ItsAMe123")  # noqa: F405
            p = Statement.createAccount("Peach", "MammaMia!")  # noqa: F405
            b = Statement.createAccount("Bowser", "M4r10SucK5")  # noqa: F405
            Statement.createFriends(m.id, p.id)  # noqa: F405
            Statement.createFriends(p.id, b.id)  # noqa: F405

    from .main import main as main_blueprint

    app.register_blueprint(main_blueprint)

    return app
```

```
# server.main
import atexit
import base64

from flask import (
    Blueprint,
    abort,
    g,
    jsonify,
    request,
)
from flask_httpauth import HTTPBasicAuth

import udp.auth

from . import Statement
from .lobbies import LobbyHandler

main = Blueprint("main", __name__)
auth = HTTPBasicAuth()
rsaKey = udp.auth.generateRsaKey()
lobbyHandler = LobbyHandler(rsaKey=rsaKey)


def quit() -> None:
    lobbyHandler.quit()


atexit.register(quit)


@auth.verify_password
def verifyPassword(username: str, password: str) -> bool:
    account = Statement.validateToken(username)  # check token
    if not account:  # if token not valid
        account = Statement.findAccount(username=username)  # check account
        if not account or not account.verifyPassword(
            password
        ):  # if account not exist or wrong password
            return False
    g.account = account
    return True


# Index
@main.route("/")
def index():
    return jsonify({})


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


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


@main.route("/auth/key")
@auth.login_required
def getKey():
    return jsonify(
        {
            "key": base64.encodebytes(g.account.private_key).decode(),
            "account-id": g.account.id,
        }
    )


```

```
@main.route("/auth/certificate")
@auth.login_required
def getCert():
    # return server certificate
    return None


@main.route("/auth/certificate/validate")
def validateCert():
    valid = False
    certificate = request.json.get("certificate")
    certificate = base64.decodebytes(certificate.encode())
    if certificate is not None:
        certificate = udp.auth.getCertificateFromDer(certificate)
        attributes = udp.auth.getUserCertificateAttributes(certificate)
        if attributes["account-id"] is not None:
            account = Statement.getAccount(attributes["account-id"])
            publicKey = udp.auth.getRsaPublicFromDer(account.public_key)
        else:
            publicKey = rsaKey.public_key()
        valid = udp.auth.validateCertificate(certificate, publicKey)
        return jsonify({"valid": valid, "account-id":
            attributes["account-id"]})
    else:
        abort(400)  # missing args


@main.route("/auth/test")
@auth.login_required
def authTest():
    return jsonify({"hello": g.account.username})


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


@main.route("/lobby/all")
@auth.login_required
def getLobbies():
    lobbies = LobbyHandler.getAll()
    games = {game.id: game.name for game in Statement.getGames()}
    data = lambda lobby: {  # noqa: E731
        "game": {"game-id": lobby.game_id, "game-name": games[lobby.game_id]},
        "size": Statement.getLobbySize(lobby.id),
        "is-full": Statement.getIsLobbyFree(lobby.id),
    }
    return jsonify({lobby.id: data(lobby) for lobby in lobbies})


@main.route("/lobby/create", methods=["POST"])
@auth.login_required
def createLobby():
    gameId = request.json.get("game-id")
    gameName = request.json.get("game-name")
    if not (gameId or gameName):  # check args
        abort(400)  # missing args
    game = None
    if gameId:  # check gameId not null
        game = Statement.getGame(gameId)
    if not game:  # check gameId null
        if gameName:  # check gameName not null
            game = Statement.findGame(gameName)
    if not game:  # check game null
        abort(404)  # no game found
    addr = _getAddr()
    lobby = lobbyHandler.createLobby(addr, game.id)
    return jsonify(
        {"lobby-id": lobby.id, "lobby-addr": lobby.getAddr(), "game-id":
            lobby.gameId}
    ), 201


def _getAddr():
    host, port = request.host.split(":")
    port = int(port)
    return (host, port)


@main.route("/lobby/")
@auth.login_required
def getLobby():
    lobbyId = request.json.get("lobby-id")
    if not lobbyId:
        abort(400)  # missing args
    lobby = lobbyHandler.getLobby(lobbyId)
    return jsonify(
        {"lobby-id": lobby.id, "lobby-addr": lobby.getAddr(), "game-id": lobby.gameId}
    )


@main.route("/lobby/members")
@auth.login_required
def getMembers():
    return jsonify(lobbyHandler.getMembers)


@main.route("/lobby/find")
@auth.login_required
def findLobby():
    gameId = request.json.get("game-id")
    gameName = request.json.get("game-name")
    if not (gameId or gameName):  # check args
        abort(400)  # missing args
    game = None
    if gameId:  # check gameId not null
        game = Statement.getGame(gameId)
    if not game:  # check gameId null
        if gameName:  # check gameName not null
            game = Statement.findGame(gameName)
    if not game:  # check game null
        abort(404)  # no game found
    lobby = lobbyHandler.findLobbies(game.id)
    lobby = lobby[0] if len(lobby) > 0 else None
    if lobby is not None:
        return jsonify(
            {
                "lobby-id": lobby.id,
                "lobby-addr": lobby.getAddr(),
                "game-id": lobby.gameId,
            }
        )
    else:
        abort(404)


@main.route("/friends/")
@auth.login_required
def getFriends():
    friends = Statement.getFriends(g.account.id)
    return jsonify(
```

```python
        {
            "friends": [
                {"id": account.id, "username": account.username} for account in friends
            ]
        }
    )


@main.route("/friends/add", methods=["POST"])
@auth.login_required
def addFriend():
    username = request.json.get("username")
    if username is None:
        abort(400)  # missing args
    account = g.account
    other = Statement.findAccount(username)
    if other is None:
        abort(404)
    Statement.createFriends(account.id, other.id)
    return jsonify(
        {
            "account": {"id": account.id, "username": account.username},
            "other": {"id": other.id, "username": other.username},
        }
    ), 201


@main.route("/friend/remove", methods=["DELETE"])
@auth.login_required
def removeFriend():
    username = request.json.get("username")
    if username is None:
        abort(400)  # missing args
    account = g.account
    other = Statement.findAccount(username)
    if other is None:
        abort(404)
    success = Statement.removeFriends(account.id, other.id)
    if success:
        return jsonify(data=[]), 204
    else:
        abort(404)


```

```
@main.route("/lobby/friends")
@auth.login_required
def getFriendLobbies():
    friends = Statement.getFriends(g.account.id)
    lobbyInfo = lambda lobby: {  # noqa: E731
        "lobby-id": lobby.id,
        "game-id": lobby.gameId,
        "game-name": Statement.getGame(lobby.gameId).name,
    }
    accountInfo = lambda account: {  # noqa: E731
        "account-id": account.id,
        "username": account.username,
    }
    lobbies = [
        {
            "account": accountInfo(account),
            "lobbies": [
                lobbyInfo(lobby) for lobby in
                    lobbyHandler.getMember(account.id)
            ],
        }
        for account in friends
        if len(lobbyHandler.getMember(account.id)) > 0
    ]
    return jsonify(lobbies)
```

```
# server.models
import datetime

import jwt
from flask import current_app
from flask_sqlalchemy import SQLAlchemy
from werkzeug.security import check_password_hash, generate_password_hash

import udp.auth as auth

db = SQLAlchemy()


# models
class Friends(db.Model):
    account_one_id = db.Column(
        db.Integer, db.ForeignKey("account.id"), primary_key=True
```

```python
    )
    account_two_id = db.Column(
        db.Integer, db.ForeignKey("account.id"), primary_key=True
    )


class Scores(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    score = db.Column(db.Integer, nullable=False)
    account_id = db.Column(db.Integer, db.ForeignKey("account.id"),
       nullable=False)
    game_id = db.Column(db.Integer, db.ForeignKey("game.id"), nullable=False)


class Account(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(255), unique=True, nullable=False)
    password = db.Column(db.String(162), nullable=False)
    private_key = db.Column(db.LargeBinary(1337))
    public_key = db.Column(db.LargeBinary(294))

    def hashPassword(self, password: str) -> None:
        self.password = generate_password_hash(password)

    def verifyPassword(self, password: str) -> bool:
        return check_password_hash(self.password, password)

    def generateToken(self, expiration: int = 600) -> str:
        data = {
            "id": self.id,
            "exp": datetime.datetime.now() +
               datetime.timedelta(seconds=expiration),
        }
        token = jwt.encode(data, current_app.config["SECRET_KEY"],
           algorithm="HS256")
        return token

    @staticmethod
    def validateToken(token: str):
        try:
            data = jwt.decode(
                token,
                current_app.config["SECRET_KEY"],
                leeway=datetime.timedelta(seconds=10),
                algorithms=["HS256"],
            )
        except:  # noqa: E722
            return None
        account = Statement.getAccount(data.get("id"))
        return account

    def generateKey(self, password: bytes) -> None:
        k = auth.generateRsaKey()
        self.private_key = auth.getDerFromRsaPrivate(k, password)
        self.public_key = auth.getDerFromRsaPublic(k.public_key())

    @staticmethod
    def decryptKey(self, key: bytes, password: bytes) ->
       auth.rsa.RSAPublicKey:
        k = auth.getRsaPrivateFromDer(key, password)
        return k


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


class Statement:
    # get
    @staticmethod
    def getGame(gameId: int) -> Game:
        return Game.query.filter_by(id=gameId).scalar()

    @staticmethod
    def getGames() -> list[Game]:
        return Game.query.all()

    @staticmethod
    def getAccount(userId: int) -> Account:
        return Account.query.filter_by(id=userId).scalar()

    @staticmethod
    def getFriends(accountId: int) -> list[Account]:
        friends = Friends.query.filter(
            (Friends.account_one_id == accountId)
            | (Friends.account_two_id == accountId)
        )
        friends = [
            friend.account_one_id
            if friend.account_one_id != accountId
            else friend.account_two_id
            for friend in friends
        ]
        friends = [Statement.getAccount(id) for id in friends]
        return friends

    # create
    @staticmethod
    def createAccount(username: str, password: str) -> Account:
        account = Account(username=username)
        account.hashPassword(password)
        account.generateKey(password.encode())
        db.session.add(account)
        db.session.commit()
        return account

    @staticmethod
    def createFriends(accountIdOne: int, accountIdTwo: int) -> Friends:
        idOne = min(accountIdOne, accountIdTwo)
        idTwo = max(accountIdOne, accountIdTwo)
        friends = Friends(account_one_id=idOne, account_two_id=idTwo)
        db.session.add(friends)
        db.session.commit()
        return friends

    @staticmethod
    def createGame(id:int, name:str, min_players:int, max_players:int) -> Game:
        game = Game(id=id, name=name, min_players=min_players, max_players=max_players)
        db.session.add(game)
        db.session.commit()
        return game

    # find
    @staticmethod
    def findAccount(username: str) -> Account | None:
        return Account.query.filter_by(username=username).scalar()

    @staticmethod
    def validateToken(token: str) -> Account | None:
        return Account.validateToken(token)
```

```python

    @staticmethod
    def findGame(gameName: str) -> Game | None:
        return Game.query.filter_by(name=gameName).scalar()

    # delete
    @staticmethod
    def removeFriends(accountIdOne: int, accountIdTwo: int) -> bool:
        idOne = min(accountIdOne, accountIdTwo)
        idTwo = max(accountIdOne, accountIdTwo)
        friends = Friends.query.filter(
            (Friends.account_one_id == idOne) & (Friends.account_two_id == idTwo)
        )
        if friends is not None:
            friends.delete()
            db.session.commit()
            return True
        else:
            return False
```

9.3.3 rps

```python
# rps.__init__
import os

import yaml


class bcolors:
    HEADER = "\033[95m"
    OKBLUE = "\033[94m"
    OKCYAN = "\033[96m"
    OKGREEN = "\033[92m"
    WARNING = "\033[93m"
    FAIL = "\033[91m"
    ENDC = "\033[0m"
    BOLD = "\033[1m"
    UNDERLINE = "\033[4m"


class Choice:
    ROCK = 0
    PAPER = 1
    SCISSORS = 2


class Outcome:
    LOOSE = 0
    WIN = 1
    DRAW = 2


QUEUE_TIMEOUT = 10

# config
CONFIG_PATH = os.path.join(os.path.dirname(__file__), "game_config.yaml")

with open(CONFIG_PATH) as f:
    config = yaml.safe_load(f)

ID = config["ID"]
NAME = config["NAME"]
MIN_PLAYERS = config["MIN_PLAYERS"]
MAX_PLAYERS = config["MAX_PLAYERS"]
```

```
# rps.__main__
import threading

from . import client, server


def runServer():
    s = server.Server((S_HOST, S_PORT))
    sT = threading.Thread(target=s.mainloop, daemon=True)
    sT.start()
    return s, sT


def runClient():
    c = client.Client((C_HOST, C_PORT), (S_HOST, S_PORT))
    return c


if __name__ == "__main__":
    import time

    from udp import C_HOST, C_PORT, S_HOST, S_PORT

    print("\n" * 4)
    s, sT = runServer()
    time.sleep(1)
    c = runClient()
    c.connect()
    time.sleep(1)
    c.isRunning = False
    time.sleep(1)
    s.isRunning = False
    time.sleep(1)
    print("END")
```

```
# rps.client
import json
from queue import Empty, Queue
from threading import Thread

import udp.error as error
from inputimeout import TimeoutOccurred, inputimeout
from udp.auth import rsa
from udp.client import Client as UdpClient
from udp.packet import Flag, lazyFlags

from . import QUEUE_TIMEOUT, Outcome, bcolors


class Client:
    isRunning: bool
    recvQueue: Queue
    score: int
    onReceiveData: None
    gameThread: Thread
    udpClient: UdpClient

    def __init__(
        self,
        addr: tuple[str, int],
        targetAddr: tuple[str, int],
        rsaKey: rsa.RSAPrivateKey|None = None,
        userId: int | str | None = None,
```

```python
        username: str | None = None,
        onReceiveData=None,
    ) -> None:
        self.isRunning = True
        self.recvQueue = Queue()
        self.score = 0
        self.onReceiveData = onReceiveData
        self.gameThread = Thread(
            name=f"{addr[1]}:Gameloop", target=self.gameloop, daemon=True
        )
        self.udpClient = UdpClient(
            addr,
            targetAddr,
            rsaKey=rsaKey,
            accountId=userId,
            username=username,
            onConnect=self.onConnect,
            onReceiveData=self.receive,
        )

    def send(self, addr: tuple[str, int], data: json) -> None:
        self.udpClient.queueDefault(
            addr, flags=lazyFlags(Flag.RELIABLE), data=self.encodeData(data)
        )

    def receive(self, addr: tuple[str, int], data: bytes):
        self.recvQueue.put((addr, self.decodeData(data)))
        if self.onReceiveData:
            self.onReceiveData(addr, data)

    @staticmethod
    def encodeData(data: dict) -> bytes:
        return json.dumps(data).encode()

    @staticmethod
    def decodeData(data: bytes) -> dict:
        return json.loads(data.decode())

    def connect(self) -> None:
        try:
            self.udpClient.connect()
        except error.PaperClipError as e:
            match e:
                case error.NoSpaceError():
                    print(
                        f"{bcolors.FAIL}Failed to join server due to
                            {error.ConnectionErrorCodes.NO_SPACE.name}:
                            {e.args[0]}{bcolors.ENDC}"
                    )
                case error.CertificateInvalidError():
                    print(
                        f"{bcolors.FAIL}Failed to join server due to
                            {error.ConnectionErrorCodes.CERTIFICATE_INVALID.name}:
                            {e.args[0]}{bcolors.ENDC}"
                    )
                case error.FinishInvalidError():
                    print(
                        f"{bcolors.FAIL}Failed to join server due to
                            {error.ConnectionErrorCodes.FINISH_INVALID.name}:
                            {e.args[0]}{bcolors.ENDC}"
                    )
                case _:
                    raise e

    def onConnect(self, addr: tuple[str, int]) -> None:
        self.gameThread.start()
        try:
            self.udpClient.mainloop(self.quit)
        except error.PaperClipError as e:
            match e:
                case error.ServerDisconnectError():
                    print(
                        f"{bcolors.FAIL}Server connection terminated due to
                            {error.DisconnectErrorCodes.SERVER_DISCONNECT.name}:
                            {e.args[0]}\nPlease wait while connection closes
                            gracefully...{bcolors.ENDC}"
                    )
                case _:
                    raise e
        if self.gameThread.is_alive():
            self.gameThread.join()
        return None

    def gameloop(self) -> None:
        print(f"{bcolors.HEADER}\n\nRock Paper Scissors{bcolors.ENDC}")
        try:
            while self.isRunning:
                choice = None
                print("Choice R[0], P[1], S[2]: ")
                while choice is None:
```

```python
                    try:
                        choice = inputimeout("", timeout=10).strip()
                        if choice == "q":
                            print(
                                f"{bcolors.FAIL}Quitting. Please wait...{bcolors.ENDC}"
                            )
                            self.isRunning = False
                            break
                        choice = int(choice)
                        if choice not in (0, 1, 2):
                            print(
                                f"{bcolors.FAIL}Invalid choice '{choice}'.{bcolors.ENDC}"
                            )
                            choice = None
                    except ValueError:
                        print(f"{bcolors.FAIL}Invalid choice.{bcolors.ENDC}")
                        choice = None
                    except KeyboardInterrupt:
                        print(f"{bcolors.FAIL}Quitting. Please wait...{bcolors.ENDC}")
                        self.isRunning = False
                        break
                    except TimeoutOccurred:
                        if not self.isRunning:
                            break
                if self.isRunning:
                    self.send(self.udpClient.targetAddr, {"choice": choice})
                    print("Waiting for other player...")
                    while self.isRunning:
                        try:
                            addr, data = self.recvQueue.get(timeout=QUEUE_TIMEOUT)
                            break
                        except Empty:
                            pass  # check still running
                    if self.isRunning:
                        match data["outcome"]:
                            case 0:
                                o = f"You {bcolors.FAIL}LOOSE{bcolors.ENDC}. "
                            case 1:
                                o = f"You {bcolors.OKGREEN}WIN{bcolors.ENDC}. "
                            case 2:
                                o = f"You {bcolors.OKCYAN}DRAW{bcolors.ENDC}. "
                            case _:
                                o = ""
                        print(
                            f"\n{o}You Picked {data['choice']}. They picked {data['otherChoice']}.\nThe score is {data['score']['score']}:{data['otherScore']['score']}."
                        )
                        if data["outcome"] == Outcome.WIN:
                            self.score += 1
                        self.recvQueue.task_done()
        finally:
            self.udpClient._quit()

    def quit(self, msg: str = "quit call", e: Exception | None = None) -> None:
        self.isRunning = False
        self.udpClient.quit(msg, e)
```

```
# game_config.yaml
NAME: "RPS"
ID: 1
MIN_PLAYERS: 2
MAX_PLAYERS: 2
```

```
# rps.server
import json
from queue import Empty, Queue
from threading import Lock

from udp.auth import rsa
from udp.packet import Flag, lazyFlags
from udp.server import Server as UdpServer

from . import MAX_PLAYERS, QUEUE_TIMEOUT, Choice, Outcome


class Server:
    isRunning: bool
    recvBuffer: Queue
    players: dict[tuple[str, int], dict[str, int]]
    playersLock: Lock
    udpServer: UdpServer
    onClientJoin: None
    onClientLeave: None
    onReceiveData: None

    def __init__(
        self,
        addr: tuple[str, int],
        rsaKey: rsa.RSAPrivateKey | None = None,
        onClientJoin=None,
        onClientLeave=None,
        onReceiveData=None,
    ):
        self.isRunning = True
        self.recvQueue = Queue()
        self.players = {}
        self.playersLock = Lock()
        self.onClientJoin = onClientJoin
        self.onClientLeave = onClientLeave
        self.onReceiveData = onReceiveData
        self.udpServer = UdpServer(
            addr,
            maxClients=MAX_PLAYERS,
            rsaKey=rsaKey,
            onClientJoin=self.playerJoin,
            onClientLeave=self.playerLeave,
            onReceiveData=self.receive,
        )

    def send(self, addr: tuple[str, int], data: dict) -> None:
        self.udpServer.queueDefault(
            addr, flags=lazyFlags(Flag.RELIABLE), data=self.encodeData(data)
        )

    def receive(self, addr: tuple[str, int], data: bytes) -> None:
        self.recvQueue.put((addr, self.decodeData(data)))
        if self.onReceiveData:
            self.onReceiveData(addr, data)

    @staticmethod
    def encodeData(data: dict) -> bytes:
        return json.dumps(data).encode()

```

```python
    @staticmethod
    def decodeData(data: bytes) -> dict:
        return json.loads(data.decode())

    @staticmethod
    def evaluateWin(choiceOne: int, choiceTwo: int) -> int:
        match choiceOne:
            case Choice.ROCK:
                match choiceTwo:
                    case Choice.ROCK:
                        return Outcome.DRAW
                    case Choice.PAPER:
                        return Outcome.LOOSE
                    case Choice.SCISSORS:
                        return Outcome.WIN
                    case _:
                        raise ValueError
            case Choice.PAPER:
                match choiceTwo:
                    case Choice.ROCK:
                        return Outcome.WIN
                    case Choice.PAPER:
                        return Outcome.DRAW
                    case Choice.SCISSORS:
                        return Outcome.LOOSE
                    case _:
                        raise ValueError
            case Choice.SCISSORS:
                match choiceTwo:
                    case Choice.ROCK:
                        return Outcome.LOOSE
                    case Choice.PAPER:
                        return Outcome.WIN
                    case Choice.SCISSORS:
                        return Outcome.DRAW
                    case _:
                        raise ValueError
            case _:
                raise ValueError

    @staticmethod
    def evaluatePlayerChoices(choices: list[tuple[tuple[str, int], int]]):
        outcomes = [
            (choices[0][0], Server.evaluateWin(choices[0][1], choices[1][1])),
            (choices[1][0], Server.evaluateWin(choices[1][1], choices[0][1])),
        ]
        return outcomes

    def getChoices(self) -> list[tuple[tuple[str, int], int]]:
        choices = {}
        while self.isRunning:
            try:
                addr, data = self.recvQueue.get(timeout=QUEUE_TIMEOUT)
                choices[addr] = data["choice"]
                if len(choices) == 2:
                    choices = [(addr, choice) for addr, choice in
                       choices.items()]
                    self.recvQueue.task_done()
                    return choices
            except Empty:
                pass  # check still running

    def playerJoin(self, addr: tuple[str, int], accountId: int) -> None:
        with self.playersLock:
            self.players[addr] = {"score": 0, "accountId": accountId}
        if self.onClientJoin:
            self.onClientJoin(addr, accountId)

    def playerLeave(self, addr: tuple[str, int], accountId: int) -> None:
        with self.playersLock:
            # TODO: submit score
            del self.players[addr]
        if self.onClientLeave:
            self.onClientLeave(addr, accountId)

    def isNotFull(self) -> bool:
        return self.udpServer.isNotFull()

    def isEmpty(self) -> bool:
        return self.udpServer.isEmpty()

    def getPlayers(self) -> dict[tuple[str, int], dict[str, int]]:
        with self.playersLock:
            return self.players.copy()

    def getPlayer(self, addr: tuple[str, int]) -> int:
        with self.playersLock:
            if addr in self.players:
                return self.players[addr]
            else:
                return None

    def setPlayer(self, addr: tuple[str, int], v: int) -> None:
        with self.playersLock:
            if addr in self.players:
                self.players[addr] = v

    def incrementPlayer(self, addr: tuple[str, int]) -> None:
        with self.playersLock:
            self.players[addr]["score"] += 1

    def getAccountId(self, addr: tuple[str, int]) -> int:
        with self.playersLock:
            return self.players[addr]["accountId"]

    def getAccountIds(self, addr: tuple[str, int]) -> list[int]:
        with self.playersLock:
            return [player["accountId"] for player in self.players.values()]

    @property
    def playerCount(self) -> int:
        with self.playersLock:
            return len(self.players)

    def mainloop(self) -> None:
        self.udpServer.startThreads()
        try:
            while self.isRunning:
                if self.playerCount == MAX_PLAYERS:
                    choices = self.getChoices()
                    outcomes = self.evaluatePlayerChoices(choices)
                    replies = {}
                    for addr, outcome in outcomes:
                        replies[addr] = {
                            "outcome": outcome,
                            "choice": [v for k, v in choices if k == addr][0],
                            "otherChoice": [v for k, v in choices if k !=
                               addr][0],
                        }
                        if outcome == Outcome.WIN:
                            self.incrementPlayer(addr)
                    scores = self.getPlayers()
                    for addr in replies:
                        replies[addr] |= {
                            "score": scores[addr],
                            "otherScore": [v for k, v in scores.items() if k != addr][
                                0
                            ],
                        }
                        self.send(addr, replies[addr])
        finally:
            self.quit()

    def quit(self) -> None:
        self.isRunning = False
        self.udpServer.quit()
```

9.3.4 client

```
# client.__init__
import os
import sys

import dotenv

from udp import logger, logging

dotenv.load_dotenv()

TCP_HOST = os.environ.get("S_HOST")
TCP_PORT = int(os.environ.get("TCP_PORT"))
C_PORT = int(os.environ.get("C_PORT"))
SERVER_URL = f"http://{TCP_HOST}:{TCP_PORT}"

offset = sys.argv[1:]
try:
    offset = int(offset[0])
except ValueError:
    offset = None
except IndexError:
    offset = None

if os.environ.get("DEBUG") is not None:
    logger.setLevel(logging.WARNING)
    while offset is None:
        try:
            offset = int(input("\noffset: "))
        except ValueError:
            print("Invalid input.")
else:
    logger.setLevel(logging.ERROR)

if offset is not None:
    C_PORT += offset
```

```
# client.__main__
import base64
import json
import time

import requests
from requests.auth import HTTPBasicAuth

import udp.auth
from rps.client import Client as RpsClient
from udp import bcolors

from . import C_PORT, SERVER_URL, TCP_HOST


class Client:
    id: int
    username: str
    password: str
    gameClient: None
    token: str
    key: udp.auth.rsa.RSAPublicKey
    auth: HTTPBasicAuth

    def __init__(self, username: str, password: str, token: str | None = None) -> None:
        self.username = username
        self.password = password
        self.gameClient = None
        self.token = (
            token if token is not None else self.getToken(self.username, self.password)
        )
        self.auth = HTTPBasicAuth(self.token, "")
        self.getKey(password.encode())
```

```python

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

    @staticmethod
    def createAccount(username: str, password: str) -> str:
        url = SERVER_URL + "/auth/register"
        headers = {"Content-Type": "application/json"}
        data = {"username": username, "password": password}
        r = requests.post(url, headers=headers, data=json.dumps(data))
        assert r.status_code == 201, r
        return r.json()["username"]

    def getKey(self, password: bytes) -> udp.auth.rsa.RSAPrivateKey:
        url = SERVER_URL + "/auth/key"
        r = requests.get(url, auth=self.auth)
        assert r.status_code == 200, r
        self.id = r.json()["account-id"]
        key = base64.decodebytes(r.json()["key"].encode())
        self.key = udp.auth.getRsaPrivateFromDer(key, password)

    # game
    def getGames(self) -> dict:
        url = SERVER_URL + "/games/"
        r = requests.get(url, auth=self.auth)
        assert r.status_code == 200, r
        return r.json()

    def getLobbies(self) -> dict:
        url = SERVER_URL + "/lobby/all"
        r = requests.get(url, auth=self.auth)
        assert r.status_code == 200, r
        return r.json()

    def createLobby(
        self, gameId: int | None = None, gameName: str | None = None
    ) -> dict:
        url = SERVER_URL + "/lobby/create"
        headers = {"Content-Type": "application/json"}
        data = {}
        if gameId:
            data["game-id"] = gameId
        elif gameName:
            data["game-name"] = gameName
        r = requests.post(url, headers=headers, data=json.dumps(data), auth=self.auth)
        assert r.status_code == 201, r
        return r.json()

    def getLobby(self, lobbyId: int) -> dict:
        url = SERVER_URL + "/lobby/"
        headers = {"Content-Type": "application/json"}
        data = {"lobby-id": lobbyId}
        r = requests.get(url, headers=headers, data=json.dumps(data), auth=self.auth)
        assert r.status_code == 200
        return r.json()

    def findLobby(self, gameId: int | None = None, gameName: str | None = None) -> dict:
        url = SERVER_URL + "/lobby/find"
        headers = {"Content-Type": "application/json"}
        data = {}
        if gameId:
            data["game-id"] = gameId
        elif gameName:
            data["game-name"] = gameName
        r = requests.get(url, headers=headers, data=json.dumps(data), auth=self.auth)
        assert r.status_code == 200, r
        return r.json()

    # friends
    def friendLobbies(self) -> dict:
        url = SERVER_URL + "/lobby/friends"
        r = requests.get(url, auth=self.auth)
        assert r.status_code == 200, r
        return r.json()

    def getFriends(self) -> dict:
        url = SERVER_URL + "/friends/"
        r = requests.get(url, auth=self.auth)
        assert r.status_code == 200, r
        return r.json()

    def addFriend(self, username: str) -> dict:
        url = SERVER_URL + "/friends/add"
        headers = {"Content-Type": "application/json"}
        data = {"username": username}
        r = requests.post(url, headers=headers, data=json.dumps(data),
            auth=self.auth)
        assert r.status_code == 201, r
        return r.json()

    def removeFriend(self, username: str) -> bool:
        url = SERVER_URL + "/friend/remove"
        headers = {"Content-Type": "application/json"}
        data = {"username": username}
        r = requests.delete(url, headers=headers, data=json.dumps(data),
            auth=self.auth)
        assert r.status_code == 204, r
        return True

    # join
    def join(self, lobbyId: int) -> None:
        print(f"\n{bcolors.WARNING}Joining Lobby '{lobbyId}'{bcolors.ENDC}")
        data = self.getLobby(lobbyId)
        if data["lobby-addr"] is not None:
            match data["game-id"]:
                case 1:
                    self.gameClient = RpsClient(
                        (TCP_HOST, C_PORT),
                        data["lobby-addr"],
                        rsaKey=self.key,
                        userId=self.id,
                        username=self.username,
                    )
                    self.gameClient.connect()
                case _:
                    raise ValueError(f"Unknown gameId {data['game-id']}")


def mainloop():
    print(f"{bcolors.HEADER}\nLobby.{bcolors.ENDC}")
    print("1. Login\n2. Register\n3. Quit")
    while True:
        option = input(": ").strip()
        match option:
            case "1":
                _login()
                break
            case "2":
                _register()
                break
            case "3":
                break
            case _:
                print(f"{bcolors.FAIL}Error: Invalid input '{option}'.{bcolors.ENDC}")


def _register(username: str | None = None, password: str | None = None):
    print(f"{bcolors.HEADER}\nRegister.{bcolors.ENDC}")
    account = None
    while account is None:
        while username is None or password is None:
            try:
                username = input("Username: ").strip()
                password = input("Password: ").strip()
            except:  # noqa: E722
                print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")
        try:
            account = Client.createAccount(username, password)
        except AssertionError:
            print(
                f"{bcolors.FAIL}Account could not be created. Please try again.{bcolors.ENDC}\n"
            )
            username = None
            password = None
    else:
        print(f"Account Created for '{account}'")
        _login(username, password)


def _login(username: str | None = None, password: str | None = None):
    print(f"{bcolors.HEADER}\nLogin.{bcolors.ENDC}")
    token = None
    while token is None:
        while username is None or password is None:
            try:
                username = input("Username: ").strip()
                password = input("Password: ").strip()
            except:  # noqa: E722
                print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")
        try:
            token = Client.getToken(username, password)
        except AssertionError:
            print(
                f"{bcolors.FAIL}Invalid login details. Please try again.{bcolors.ENDC}\n"
            )
            username = None
            password = None
    else:
        client = Client(username, password, token)
        _menu(client)


def _menu(client):
    isRunning = True
    while isRunning:
        print(f"\n{bcolors.HEADER}Main Menu{bcolors.ENDC}")
        print(f"{bcolors.OKGREEN}Hello {client.username}.{bcolors.ENDC}")
        while True:
            print(
                "\n1. Manage friends\n2. See available games\n3. Start or join a lobby\n4. Quit"
            )
            option = input(": ").strip()
            match option:
                case "1":
                    _friends(client)
                    break
                case "2":
                    _game(client)
                    break
                case "3":
                    _lobby(client)
                    break
                case "4":
                    isRunning = False
                    break
                case _:
                    print(
                        f"{bcolors.FAIL}Error: Invalid input '{option}'.{bcolors.ENDC}"
                    )


```

```python
def _friends(client: Client):
    while True:
        print(f"{bcolors.HEADER}\nFriends.{bcolors.ENDC}")
        friends = client.getFriends()
        friends = "\n\t".join(
            [
                f"{i+1}. {friend['username']}"
                for i, friend in enumerate(friends["friends"])
            ]
        )
        print(f"Friend list: \n\t{friends}")
        print("\n1. Add New Friend\n2. Remove Friend\n3. Return to Main Menu")
        while True:
            option = input(": ").strip()
            match option:
                case "1" | "2":
                    username = input("\nUsername: ").strip()
                    match option:
                        case "1":
                            try:
                                client.addFriend(username)
                                print(
                                    f"\n{bcolors.OKGREEN}Account '{username}' added as friend{bcolors.ENDC}"
                                )
                                break
                            except AssertionError:
                                print(
                                    f"\n{bcolors.FAIL}Error: No such account with username '{username}'.{bcolors.ENDC}"
                                )
                        case "2":
                            try:
                                client.removeFriend(username)
                                print(
                                    f"\n{bcolors.OKGREEN}Account '{username}' removed as friend{bcolors.ENDC}"
                                )
                                break
                            except AssertionError:
                                print(
                                    f"\n{bcolors.FAIL}Error: No such account with username '{username}' in friend list.{bcolors.ENDC}"
                                )
                case "3":
                    return None
                case _:
                    print(
                        f"{bcolors.FAIL}Error: Invalid input '{option}'.{bcolors.ENDC}"
                    )


def _game(client: Client):
    while True:
        print(f"{bcolors.HEADER}\nGames{bcolors.ENDC}")
        availableGames = client.getGames()
        availableGames = "\n\t".join(
            [f"{id}. {game}" for id, game in availableGames.items()]
        )
        print(f"Available Games: \n\t{availableGames}")
        input("\nPress enter to return to main menu: ")
        return None


def _lobby(client: Client):
    while True:
        print(f"{bcolors.HEADER}\nLobby.{bcolors.ENDC}")
        print(
            "\n1. Matchmaking\n2. See Friends' Lobbies\n3. Join Lobby\n4. Create Lobby\n5. Return to Main Menu"
        )
        while True:
            option = input(": ").strip()
            match option:
                case "1":
                    _matchmaking(client)
                    break
                case "2":
                    _friendsLobbies(client)
                    break
                case "3":
                    _joinLobby(client)
                    break
                case "4":
                    _createLobby(client)
                    break
                case "5":
                    return None
                case _:
                    print(
                        f"{bcolors.FAIL}Error: Invalid input
                            '{option}'.{bcolors.ENDC}"
                    )


def _matchmaking(client: Client):
    print(f"{bcolors.HEADER}\nMatchmaking.{bcolors.ENDC}")
    game = _gameInput(client)
    if game is None:
        return None
    try:
        lobby = client.findLobby(gameName=game)
    except AssertionError:
        lobby = client.createLobby(gameName=game)
    time.sleep(1)
    client.join(lobby["lobby-id"])
    return None


def _friendsLobbies(client: Client):
    print(f"{bcolors.HEADER}\nFriends' Lobbies.{bcolors.ENDC}")
    lobbies = client.friendLobbies()
    lobbiesInfo = lambda lobbies: "\n\t\t".join(
        [
            f"{bcolors.OKCYAN}{lobby['lobby-id']}{bcolors.ENDC}.
                {lobby['game-name']}"
            for lobby in lobbies
        ]
    )  # noqa: E731
    lobbies = "\n\t".join(
        [
            f"\n\t{i+1}.
                {account['account']['username']}:\n\t\t{lobbiesInfo(account['lobbies'])}"
            for i, account in enumerate(lobbies)
        ]
    )
    print(f"\nLobbies:{lobbies}")
    print(
        f"Input {bcolors.OKCYAN}Lobby Id{bcolors.ENDC} to Join Friend or
            Press Enter to Return to Menu."
    )
    while True:
```

```python
        option = input(": ").strip()
        if option == "":
            return None
        else:
            try:
                option = int(option)
                client.join(option)
                return None
            except ValueError:
                print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")


def _joinLobby(client: Client):
    print(f"{bcolors.HEADER}\nJoin Lobby.{bcolors.ENDC}")
    lobbyId = None
    while lobbyId is None:
        try:
            lobbyId = input("\nLobby Id: ").strip()
            if lobbyId == "":
                return None
            else:
                lobbyId = int(lobbyId)
        except ValueError:
            print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")
    try:
        client.join(lobbyId)
        return None
    except:  # noqa: E722
        return None


def _createLobby(client: Client):
    print(f"{bcolors.HEADER}\nCreate Lobby.{bcolors.ENDC}")
    game = _gameInput(client)
    while True:
        if game is None:
            return None
        try:
            lobby = client.createLobby(gameName=game)
            client.join(lobby["lobby-id"])
            return None
        except AssertionError:
            print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")


```

```python
def _gameInput(client: Client) -> str:
    availableGames = client.getGames()
    games = "\n\t".join([f"{id}. {game}" for id, game in availableGames.items()])
    print(f"\nAvailable Games: \n\t{games}")
    game = None
    while game is None or game.lower() not in map(
        lambda x: x.lower(), availableGames.values()
    ):
        try:
            game = input("Game: ").strip()
            if game == "":
                return None
        except:  # noqa: E722
            print(f"{bcolors.FAIL}Error: Invalid input.{bcolors.ENDC}")
    return game


if __name__ == "__main__":
    mainloop()
```

```
# .env
# udp
S_HOST=127.0.0.1
S_PORT=2024
C_HOST=127.0.0.1
C_PORT=2025
## node
SOCKET_BUFFER_SIZE = 1024
SEND_SLEEP_TIME = 0.1
QUEUE_TIMEOUT = 10
SOCKET_TIMEOUT = 20
## server
HEARTBEAT_MAX_TIME = 120
HEARTBEAT_MIN_TIME = 30
MAX_CLIENTS
## auth
ORG_NAME = Paperclip
COMMON_NAME = 127.0.0.1
## utils
MAX_FRAGMENT_SIZE = 988

# client
```

```
TCP_PORT = 5000

# app
FLASK_APP = server
PRUNE_TIME = 58
SECRET_KEY = MyVerySecretKey
SQLALCHEMY_DATABASE_URI = mysql://root:root@localhost:3306/paperclip

# debug
DEBUG = True
```

```
cryptography==42.0.5
Flask==3.0.2
Flask-HTTPAuth==4.8.0
Flask-SQLAlchemy==3.1.1
SQLAlchemy-Utils==0.41.2
mysqlclient==2.2.4
requests==2.31.0
PyJWT==2.8.0
pytest==8.1.1
python-dotenv==1.0.1
PyYAML==6.0.1
```

```
# test_udp
import os
import threading
from random import choice, randint

from udp import C_HOST, C_PORT, auth, error, node, packet, utils


## node
def testNodeSequenceIdLock():
    n = node.Node((C_HOST, C_PORT))

    def test():
        for _ in range(100000):
            n.incrementSequenceId(n.addr)

    threads = [threading.Thread(target=test) for _ in range(10)]
    for t in threads:
        t.start()
    for t in threads:
        t.join()
    assert n.sequenceId == 16960, n.sequenceId


# error
def testErrorCode():
    major = choice([i for i in error.Major])
    minor = error.getMinor(major, randint(0, 2))
    mm = (major, minor)
    e = error.getError(*mm)()
    c = error.getErrorCode(e.__class__)
    assert mm == c, (mm, e, c)


def testErrorPacket():
    h = genRandAttr(packet.Type.ERROR)
    p = packet.ErrorPacket(*h)
    p.data = b"This is a test error"
    p.major = randint(1, 3)
    match p.major:
        case error.Major.CONNECTION:
            p.minor = randint(0, 3)
        case error.Major.DISCONNECT:
            p.minor = randint(0, 2)
        case error.Major.PACKET:
            p.minor = randint(0, 9)
        case _:
            p.minor = 0
    eP = p.pack(p)
    dP = packet.unpack(eP)
    assert p == dP, (p, eP, dP)


# Heartbeat
def testHeartbeatPacket():
    h = genRandAttr(packet.Type.HEARTBEAT)
    p = packet.HeartbeatPacket(*h)
    p.heartbeat = True
    eP = p.pack(p)
    dP = packet.unpack(eP)
    assert p == dP, (p, eP, dP)


```

```
# frag
def testDefrag():
    h = genRandAttr()
    data = os.urandom(16)
    p = packet.Packet(*h)
    p.flags[packet.Flag.FRAG.value] = 0
    p.fragment_id = None
    p.fragment_number = None
    p.data = data
    fP = p.fragment()
    dP = fP[0].defragment(fP)
    assert p == dP, (p, fP, dP)


## utils
def testDataCompress(d=os.urandom(16)):
    cD = utils.compressData(d)
    dD = utils.decompressData(cD)
    assert d == dD, (d, cD, dD)


## encrypt
def testPacketEncryption():
    h = genRandAttr()
    p = packet.Packet(*h)
    p.flags[packet.Flag.ENCRYPTED.value] = 1
    d = b"Hello World"
    p.data = d
    localKey = auth.generateEcKey()
    peerKey = auth.generateEcKey()
    localSessionKey = auth.generateSessionKey(localKey, peerKey.public_key())
    peerSessionKey = auth.generateSessionKey(peerKey, localKey.public_key())
    p.encryptData(localSessionKey)
    # print(p.data)
    p.decryptData(peerSessionKey)
    # print(p.data)
    assert d == p.data, (d, p.data)


## auth
def sessionKey():
    localKey = auth.generateEcKey()
    peerKey = auth.generateEcKey()
    localSessionKey = auth.generateSessionKey(localKey, peerKey.public_key())
    peerSessionKey = auth.generateSessionKey(peerKey, localKey.public_key())
    assert localSessionKey == peerSessionKey


def encryptDecrypt(inputText=b"Hello World"):
    localKey = auth.generateEcKey()
    peerKey = auth.generateEcKey()
    sessionKey = auth.generateSessionKey(localKey, peerKey.public_key())
    #
    localCipher, iv = auth.generateCipher(sessionKey)
    cipherText = auth.encryptBytes(localCipher, inputText)
    #
    peerCipher, _ = auth.generateCipher(sessionKey, iv)
    outputText = auth.decryptBytes(peerCipher, cipherText)
    #
    assert inputText == outputText, (inputText, outputText)


## packet
def genRandAttr(t=packet.Type.DEFAULT):
    v, pT, sId = randint(0, 1), t, randint(0, 2**packet.SEQUENCE_ID_SIZE - 1)
    f = [0 for _ in range(packet.FLAGS_SIZE)]
    if randint(0, 1):
        f[packet.Flag.FRAG.value] = 1
        fId, fNum = (
            randint(0, 2**packet.FRAGMENT_ID_SIZE - 1),
            randint(0, 2**packet.FRAGMENT_NUM_SIZE - 1),
        )
    else:
        fId, fNum = None, None
    if randint(0, 1):
        f[packet.Flag.ENCRYPTED.value] = 1
        # iv = randint(0, 2**INIT_VECTOR_SIZE-1)
        iv = auth.generateInitVector()
    else:
        iv = None
    if randint(0, 1):
        f[packet.Flag.CHECKSUM.value] = 1
        c = randint(0, 2**packet.CHECKSUM_SIZE - 1)
    else:
        c = None
    h = (v, pT, f, sId, fId, fNum, iv, c)
    return h


def testAuth():
```

```python
    pK, c = (
        auth.generateEcKey().public_key(),
        auth.generateUserCertificate(auth.generateRsaKey()),
    )
    pKS, cS = (
        packet.AuthPacket.getPublicKeyBytesSize(pK),
        packet.AuthPacket.getCertificateByteSize(c),
    )
    h = (*genRandAttr(packet.Type.AUTH), pKS, pK, cS, c)
    # static test
    eH = packet.AuthPacket.encodeHeader(*h)
    dH = packet.AuthPacket.decodeHeader(eH)[:-1]
    assert h == dH, (h, eH, dH)
    # class tests
    p = packet.AuthPacket(*h)
    eP = p.pack(p)
    dP = p.unpack(eP)
    assert p == dP, (p, eP, dP)


def testAck():
    # header
    aId, aB = (
        randint(0, 2**packet.ACK_ID_SIZE - 1),
        [randint(0, 1) for _ in range(packet.ACK_BITS_SIZE)],
    )
    h = (*genRandAttr(packet.Type.ACK), aId, aB)
    # static test
    eH = packet.AckPacket.encodeHeader(*h)
    dH = packet.AckPacket.decodeHeader(eH)[:-1]
    assert h == dH, (h, eH, dH)
    # class tests
    p = packet.AckPacket(*h)
    eP = p.pack(p)
    dP = p.unpack(eP)
    assert p == dP, (p, eP, dP)


def testAckBits():
    aId, aB = (
        randint(0, 2**packet.ACK_ID_SIZE - 1),
        [randint(0, 1) for _ in range(packet.ACK_BITS_SIZE)],
    )
    eAId, eAB = packet.AckPacket.encodeAckId(aId),
       packet.AckPacket.encodeAckBits(aB)
    dAId, dAB = packet.AckPacket.decodeAckId(eAId),
       packet.AckPacket.decodeAckBits(eAB)
    assert (aId, aB) == (dAId, dAB), ((aId, aB), (eAId, eAB), (dAId, dAB))


def testDefault():
    # header
    h = genRandAttr()
    # static test
    eH = packet.Packet.encodeHeader(*h)
    dH = packet.Packet.decodeHeader(eH)[:-1]
    assert h == dH, (h, eH, dH)
    # class tests
    p = packet.Packet(*h)
    eP = p.pack(p)
    dP = p.unpack(eP)
    assert p == dP, (p, eP, dP)


def testChecksum():
    # checksum
    c = randint(0, 2**packet.CHECKSUM_SIZE - 1)
    eC = packet.Packet.encodeChecksum(c)
    dC = packet.Packet.decodeChecksum(eC)
    assert c == dC, (c, eC, dC)


def testInitVector():
    # init vector
    iv = randint(0, 2**packet.INIT_VECTOR_SIZE - 1)
    eIv = packet.Packet.encodeInitVector(iv)
    dIv = packet.Packet.decodeInitVector(eIv)
    assert iv == dIv, (iv, eIv, dIv)


def testFrag():
    # frag
    fId, fN = (
        randint(0, 2**packet.FRAGMENT_ID_SIZE - 1),
        randint(0, 2**packet.FRAGMENT_NUM_SIZE - 1),
    )
    eFId, eFN = (
        packet.Packet.encodeFragmentId(fId),
        packet.Packet.encodeFragmentNumber(fN),
    )
```

```python
    dFId, dFN = (
        packet.Packet.decodeFragmentId(eFId),
        packet.Packet.decodeFragmentNumber(eFN),
    )
    assert (fId, fN) == (dFId, dFN), ((fId, fN), (eFId + eFN), (dFId, dFN))


def testFlags():
    # flags
    f = [randint(0, 1) for _ in range(packet.FLAGS_SIZE)]
    eF = packet.Packet.encodeFlags(f)
    dF = packet.Packet.decodeFlags(eF)
    assert f == dF, (f, eF, dF)


def testVersionType():
    # version type
    v, pT = (
        randint(0, 2**packet.VERSION_SIZE - 1),
        packet.Type(randint(0, max(t.value for t in packet.Type))),
    )
    eVt = packet.Packet.encodeVersionType(v, pT)
    dVt = packet.Packet.decodeVersionType(eVt)
    assert (v, pT) == dVt, ((v, pT), eVt, dVt)
```

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

```python
from .inputimeout import inputimeout, TimeoutOccurred  # noqa
from .__version__ import (   # noqa
    __version__, __author__, __author_email__, __copyright__, __license__,
    __description__, __title__, __url__,
)
```

```python
__title__ = 'inputimeout'
__description__ = 'Multi platform standard input with timeout'
__url__ = 'http://github.com/johejo/inutimeout'
__version__ = '1.0.4'
__author__ = 'Mitsuo Heijo'
__author_email__ = 'mitsuo_h@outlook.com'
```

```
__license__ = 'MIT'
__copyright__ = 'Copyright 2018 Mitsuo Heijo'
```

```
# Modified by @HarryWhitehorn on 2024/04/27:
# - Modified win_inputimeout to prevent automatically appending a newline

import sys

DEFAULT_TIMEOUT = 30.0
INTERVAL = 0.05

SP = ' '
CR = '\r'
LF = '\n'
CRLF = CR + LF


class TimeoutOccurred(Exception):
    pass


def echo(string):
    sys.stdout.write(string)
    sys.stdout.flush()


def posix_inputimeout(prompt='', timeout=DEFAULT_TIMEOUT):
    echo(prompt)
    sel = selectors.DefaultSelector()
    sel.register(sys.stdin, selectors.EVENT_READ)
    events = sel.select(timeout)

    if events:
        key, _ = events[0]
        return key.fileobj.readline().rstrip(LF)
    else:
        echo(LF)
        termios.tcflush(sys.stdin, termios.TCIFLUSH)
        raise TimeoutOccurred


def win_inputimeout(prompt='', timeout=DEFAULT_TIMEOUT, newline=False):
    echo(prompt)
    begin = time.monotonic()
    end = begin + timeout
    line = ''

    while time.monotonic() < end:
        if msvcrt.kbhit():
            c = msvcrt.getwche()
            if c in (CR, LF):
                echo(CRLF)
                return line
            if c == '\003':
                raise KeyboardInterrupt
            if c == '\b':
                line = line[:-1]
                cover = SP * len(prompt + line + SP)
                echo(''.join([CR, cover, CR, prompt, line]))
            else:
                line += c
        time.sleep(INTERVAL)

    if newline:
        echo(CRLF)
    raise TimeoutOccurred


try:
    import msvcrt

except ImportError:
    import selectors
    import termios

    inputimeout = posix_inputimeout

else:
    import time

    inputimeout = win_inputimeout
```

```
MIT License

Copyright (c) 2017 Mitsuo Heijo

```

```
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
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 receival. 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 receival 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 $[ID_{-1}, ID_{-2}, ID_{-3}, ..., ID_{-17}]$ 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	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

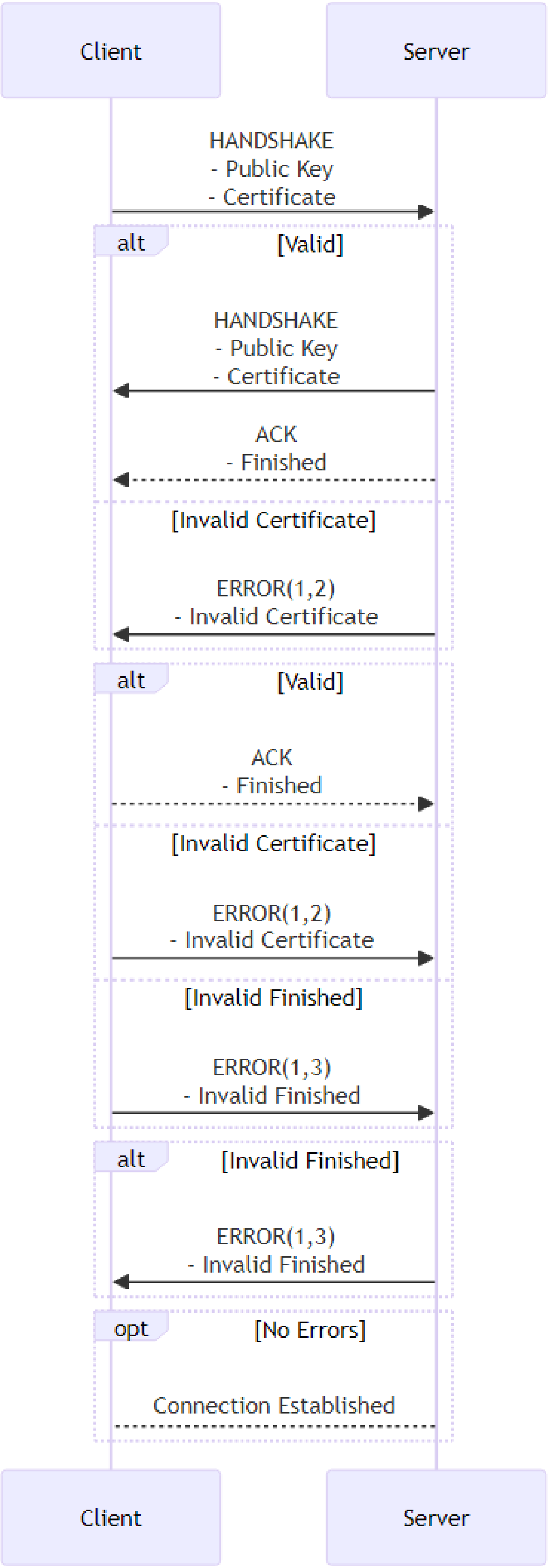
This field allows for additional information about the error to be shared. This used for logging purposes but can also be used for displaying *user-friendly* error messages to a client.

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.


Client
Server
HANDSHAKE
- Public Key
- Certificate
alt
[Valid]
HANDSHAKE
- Public Key
- Certificate
ACK
- Finished
[Invalid Certificate]
ERROR(1,2)
- Invalid Certificate
alt
[Valid]
ACK
- Finished
[Invalid Certificate]
ERROR(1,2)
- Invalid Certificate
[Invalid Finished]
ERROR(1,3)
- Invalid Finished
alt
[Invalid Finished]
ERROR(1,3)
- Invalid Finished
opt
[No Errors]
Connection Established
Client
Server

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 Cetficiate	/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 registed as a Friend
Friends						
🟩	Get Friends	/friends	GET		list[Friend]	Return a list of all Accounts registed 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

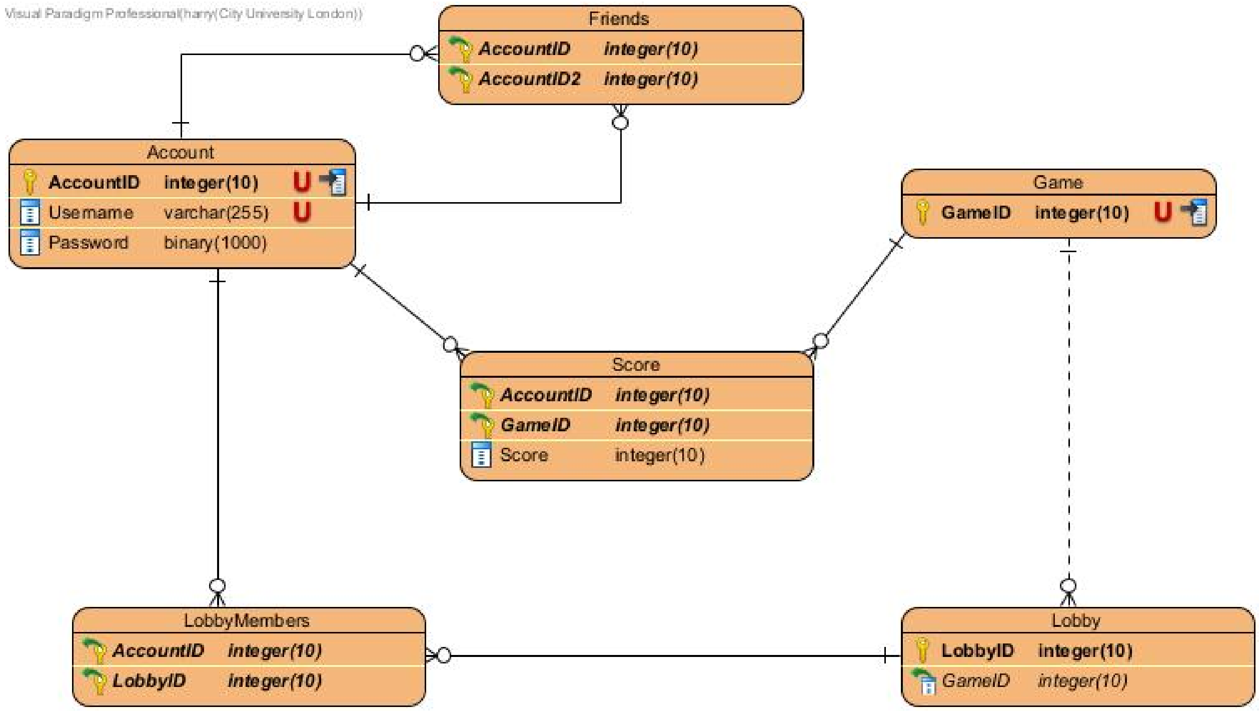


Figure 10: Database Models ERD