## Contents

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

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

## **Paperclip**

#### Abstract

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

## 1 Introduction

## 1.1 Problem Description

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

## 1.2 Objectives

## 1.2.1 Primary Objectives

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

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

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

#### 1.3 Beneficiaries

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

## 1.4 Assumptions and limitations

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

## 2 Output Summary

Name	udp
Type	Bython Package
Size	1 yunui 1 acaage 2031 lines (after formatting)
Credit	2090 lines (after formatting)
Description	Implimentation of custom udp protocol with client and server
Usage	Implimentation of custom tudp protocol with cient and server  Base objects for game developer to build on for implimation into their game.
Link	base objects for game developer to build on for implimation into their game.
LIIIK	
NT.	
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 authentification 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 input imeout which was modified to disable the automatic appending of new lines on a timeout.
Description	A cmd line client example for end user use. Handles 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	To mee (meer to meems)
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.
Coage	- The server is used by the server package when creating a game session.
Link	The out of the document of the out of parameter of the out of the
- Dillin	
Name	tests
Type	Python tests
Size	Tymor cases 264 lines (after formatting)
Credit	
Description	A pytest script to test that implimented features are working as exspected.
Usage	A pyress script to test that implaneated reactives are working as exspected. Used to validate functionality.
Link	Over to variance functionality.
Lilik	

## 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.

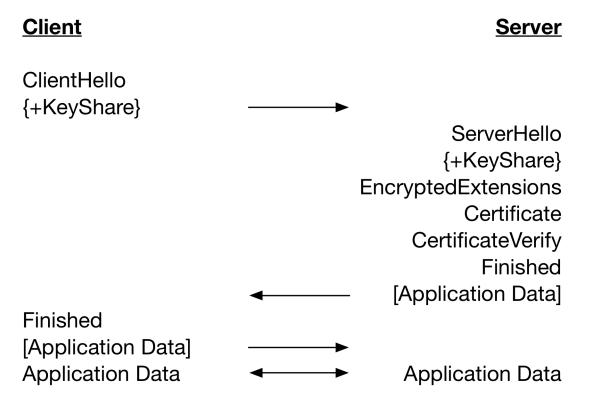


Figure 1: A example of a TLS 1.3 full handshake including a server certificate (wolfSSL, 2019)

**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 CONSTs defined across the project were consolidated into a .env file. This constance are then loaded at run-time using the dotenv package. This provided structure to the project and allowed for easier control over the various variables.
- **4.4.4.2 Logging** Logging using the logging model was implemented across the package. The logger was set to output to both the console as well as a paperclip.log file. This outputs were given different 'levels' to avoid cluttering the console output while retaining all generated outputs in the log file.
- **4.4.4.3 Database Models** The database modules defined in the ERD were implemented as SQLAlchemy db.models allowing the Server to initiate the database with the appropriate tables on start-up. Various changes were made between the design and final implementation to match the projects new requirements. These are outlined in the *Results* section.
- **4.4.4.4 RESTful Server** The TCP RESTful API Server was implemented using as a Flask app. Authentication using HTTPBasicAuth is implemented allow for either a username and password or a session key to be used.

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

- **4.4.4.5** Certificates and Handshake The udp handshake is amended to use the RESTful server as an authenticator for certificates. Additionally, the udp.auth method to generate certificates is expanded to accept and embed an Account.id and an Account.username in the certificate fields.
- **4.4.4.6 RPS Demo** A turn-based game demo (Rock, Paper, Scissors) was created containing a game Server and Client. The Server is responsible for evaluating each turn sending the results to the Clients. The Client is responsible for taking a player input and sending it to the Server the Client then displays the results received.

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

#### 4.5 Tests

The pytest module was used to define serval test.

#### 4.6 Reused Code and Tutorials

The client package makes use of 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 perticularly the use of ack\_bits in an ACK package.

## 5 Results

## 5.3 Design

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

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

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

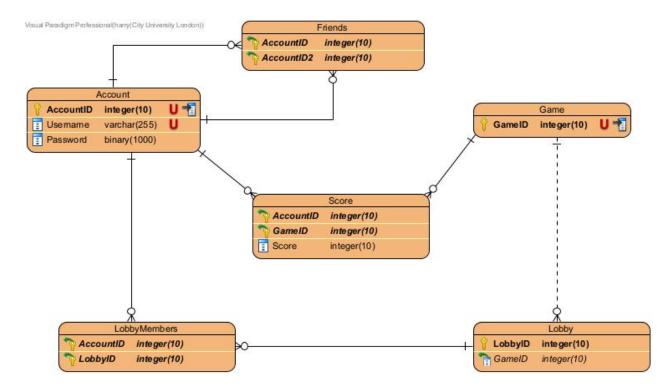


Figure 2: Database Models ERD

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

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

#### 5.4 Implementation

#### 5.4.1 Iteration 1

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

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

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

**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.

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.

#### 5.4.1.2.2 Receiving Data

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

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

#### 5.4.1.2.3 UDP Client

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

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

#### 5.4.1.2.4 UDP Server

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

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

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

```
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)

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.

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

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

#### 5.4.1.4.2 Inbound Thread

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

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

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

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

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

```
1 def listen(self) -> None:
      while self.isRunning.is set():
 3
           p, addr = self.receivePacket()
           if p is not None and addr is not None:
 4
               if self.checkClientExists(addr): # client exists
5
                   if self.getHandshake(addr): # client handshake complete =>
 6
                      allow all packet types
 7
                       self.receive(p, addr)
 8
                   else:
 9
                       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)
10
11
               else:
```

#### 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.

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

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

```
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).

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

#### 5.4.1.4.4 Server Clients

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

```
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.

#### 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.

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.

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

```
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.

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

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

## 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.

```
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.

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

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

```
1 def generateUserCertificate(key) -> x509.Certificate:
      name = [
 3
          x509.NameAttribute(NameOID.ORGANIZATION NAME, ORG NAME), # ORG_NAME
              defined as const e.g. "Paperclip"
           x509.NameAttribute(NameOID.COMMON_NAME, COMMON_NAME), # COMMON_NAME
4
              defined as const e.g. "127.0.0.1"
5
 6
      subject = issuer = x509.Name(name) # self signed
      cert = (
 7
 8
          x509.CertificateBuilder()
           .subject name(subject)
9
           .issuer name(issuer)
10
           .public_key(key.public_key())
11
12
           .serial number(x509.random serial number())
           .not valid before(datetime.datetime.now(datetime.timezone.utc))
13
           .not valid after(
14
               datetime.datetime.now(datetime.timezone.utc) +
15
                  datetime.timedelta(days=1) # valid for one day
16
           .add extension(
17
               x509.SubjectAlternativeName([x509.DNSName("localhost")]), # self
18
                  signed
               critical=False,
19
20
           .sign(key, hashes.SHA256())
21
```

```
22 )
23 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.

#### 5.4.2.2.2 Receiving an AUTH Packet

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

```
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.

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

#### 5.4.2.3.1 Client Handshake

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

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

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

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

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

## 5.4.2.3.2 Server Handshake

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

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

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

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

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

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

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

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

#### 5.4.2.4.1 ENCRYPT

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

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

#### 5.4.2.4.1.1 Encryption

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

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

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

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

```
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.

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

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

```
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.

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

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

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

## 5.4.2.4.2.2 Decompression

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

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

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

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

#### 5.4.2.4.3 CHECKSUM

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

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

## 5.4.2.4.3.1 Setting a Checksum

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

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

```
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.

#### 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.

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

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

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

The utils.fragmentData method splits the data into a list of bytes, splitting the data into fragments with a max size MAX\_FRAGMENT\_SIZE. The MAX\_FRAGMENT\_SIZE is set to 988 to keep the total packet size under 1024 (SOCKET\_BUFFER\_SIZE) when including the maximum theoretical header size.

## 5.4.2.4.4.2 Defragmentation

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

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

```
13 else:
14 return False
```

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

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

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

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

## 5.4.2.4.5 Automatic Handling

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

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

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

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

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

#### 5.4.3 Iteration 3

5.4.3.1 ACK Bits and Rolling Reset The Node class utilizes 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.

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

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

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

```
newDiff = (currentSeqId - newSeqId) % (2**16)
if newDiff < currentDiff:
    return currentSeqId
else:
return newSeqId</pre>
```

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

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

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

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

**5.4.3.2 HEARTBEAT Packets** When a Node receives a packet it updates 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.

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

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

The heartbeatThread uses the heartbeat method.

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

```
1 def heartbeat(self) -> None:
 2
           while self.isRunning.is_set():
               time.sleep(HEARTBEAT MIN TIME)
 3
               with self.clientsLock:
 4
                   clients = [k for k in self.clients.keys()]
5
               for clientAddr in clients:
 6
 7
                   heartbeat = self.getHeartbeat(clientAddr)
                   delta = (datetime.now() - heartbeat).seconds
 8
                   if delta > HEARTBEAT MAX TIME:
9
                       self.removeClient(
10
                           clientAddr.
11
12
                           debugStr=f"due to heartbeat timeout (last contact was
                               {heartbeat})",
                       )
13
                   elif delta > HEARTBEAT MIN TIME:
14
                       self.queueHeartbeat(clientAddr, heartbeat=False)
15
1 def removeClient(self, clientAddr: tuple[str, int], debugStr="") -> None:
 2
      if self.checkClientExists(clientAddr):
           cId = self.getClientId(clientAddr)
 3
           with self.clientsLock:
 4
               self.clients[clientAddr].isRunning.clear()
 5
               del self.clients[clientAddr]
 6
               if self.onClientLeave:
 7
                   self.onClientLeave(clientAddr, cId)
 8
```

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

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

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

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

## **5.4.3.4 ERROR Packets** 5.4.3.4.1 Exceptions

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

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

The three Major error types then 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.

```
1 # connection
 2 class ConnectionErrorCodes(Minor):
      CONNECTION = O
4
      NO SPACE = 1
      CERTIFICATE_INVALID = 2
 5
      FINISH INVALID = 3
 6
 8 class ConnectionError(PaperClipError): """Handshake connection could not be
      finished"""
10 class NoSpaceError(ConnectionError): """Server has insufficient space to
      accept new clients"""
12 class CertificateInvalidError(ConnectionError): """Certificate is invalid /
      can not be validated"""
13
14 class FinishInvalidError(ConnectionError): """Finish is invalid"""
15
16 connectionErrors = {
      ConnectionErrorCodes.CONNECTION: ConnectionError,
17
      ConnectionErrorCodes.NO_SPACE: NoSpaceError,
18
      ConnectionErrorCodes.CERTIFICATE INVALID: CertificateInvalidError,
19
      ConnectionErrorCodes.FINISH INVALID: FinishInvalidError,
20
21 }
22
23 def getConnectionError(minor: ConnectionErrorCodes | int) -> ConnectionError:
24
      try:
          minor = minor if isinstance(minor, Minor) else
25
              ConnectionErrorCodes(minor)
           if minor in connectionErrors:
26
               return _connectionErrors[minor]
27
           else:
28
              return PaperClipError
29
      except ValueError:
30
31
           return PaperClipError
32
33 def getConnectionCode(error: ConnectionError) -> ConnectionErrorCodes:
34
      try:
          return list( connectionErrors.keys())[
35
               list(_connectionErrors.values()).index(error)
36
37
      except ValueError:
38
39
           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.

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

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

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

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

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

### 5.4.3.4.2 Sending an ERROR Packet

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

```
1 def queueError(
      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:
3
      sId = self.getSequenceId(addr)
4
      p = packet.ErrorPacket(
 5
           sequence_id=sId if sId is not None else 0,
 6
 7
           flags=flags,
 8
          major=major,
9
           minor=minor,
10
           data=data,
11
      if sId is not None:
12
           self.incrementSequenceId(addr)
13
14
      self.queuePacket(addr, p)
```

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

```
1 def receivePacket(self,) -> tuple[packet.Packet, tuple[str, int]] |
     tuple[None, None]:
     data, addr = self.socket.recvfrom(SOCKET_BUFFER_SIZE)
2
3
4
          p = packet.unpack(data) # unpacking can yield a PacketError
          return p, addr
5
      except error.PacketError as e:
6
          major, minor = error.getErrorCod(e)
7
          self.queueError(addr, major, minor)
8
9
          return None, None
1 def receive(self, p: packet.Packet, addr: tuple[str, int]) ->
     tuple[packet.Packet, tuple[str, int]] | None:
2
     if p is not None:
          if self.handleFlags(p, addr):
3
              match p.packet type:
4
                  # packet type cases omitted for clarity
5
                  case _: # unknown packet type
6
7
                      self.queueError(
8
                          addr,
9
                          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.

```
1 case packet.Type.ERROR:
2
      try:
3
          return self.receiveError(p, addr)
      except error.PaperClipError as e:
4
          self.handleError(p, addr, e)
1 def receiveError(self, p: packet.ErrorPacket, addr: tuple[str, int]) -> None:
          raise error.getError(p.major, p.minor)(p.data)
1 def handleError(self, p: packet.ErrorPacket, addr: tuple[str, int], e:
      error.PaperClipError) -> None:
2
      match e:
          case error.ConnectionError():
3
               self.handleConnectionError(p, addr, e)
4
          case error.DisconnectError():
5
               self.handleDisconnectError(p, addr, e)
6
          case error.PacketError():
8
               self.handlePacketError(p, addr, e)
9
          case :
10
              raise e
```

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

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

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

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

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

**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.

```
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.

```
1 def handleDisconnectError(
2          self, p: packet.ErrorPacket, addr: tuple[str, int], e:
                error.DisconnectError
3          ) -> 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.

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

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

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

### 5.4.4 Iteration 4

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

```
1 # .env
 2
3 # udp
4 S HOST=127.0.0.1
5 S_PORT=2024
6 C HOST=127.0.0.1
7 C PORT=2025
8 ## node
9 SOCKET_BUFFER_SIZE = 1024
10 SEND SLEEP TIME = 0.1
11 QUEUE TIMEOUT = 10
12 SOCKET TIMEOUT = 20
13 ## server
14 HEARTBEAT MAX TIME = 120
15 HEARTBEAT MIN TIME = 30
16 MAX_CLIENTS
17 ## auth
18 ORG NAME = Paperclip
19 COMMON NAME = 127.0.0.1
20 ## utils
21 MAX_FRAGMENT_SIZE = 988
22
23 # client
24 \text{ TCP\_PORT} = 5000
25
26 # app
27 FLASK APP = server
28 PRUNE TIME = 58
29 SECRET_KEY = MyVerySecretKey
30 SQLALCHEMY DATABASE URI = mysql://root:root@localhost:3306/paperclip
31
32 # debug
33 DEBUG = True
```

The variables can then be loaded from the os.environ by first calling dotenv.load\_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)

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

```
6 S PORT = int(os.environ.get("S PORT"))
7 C_HOST = os.environ.get("C_HOST")
8 C PORT = int(os.environ.get("C PORT"))
9 # node
10 SOCKET_BUFFER_SIZE = int(os.environ.get("SOCKET_BUFFER_SIZE"))
11 SEND_SLEEP_TIME = float(os.environ.get("SEND_SLEEP_TIME"))
12 QUEUE TIMEOUT = int(os.environ.get("QUEUE TIMEOUT"))
13 SOCKET TIMEOUT = int(os.environ.get("SOCKET TIMEOUT"))
14 # server
15 HEARTBEAT MAX TIME = int(os.environ.get("HEARTBEAT MAX TIME"))
16 HEARTBEAT MIN TIME = int(os.environ.get("HEARTBEAT MIN TIME"))
17 \text{ MAX CLIENTS} = (
      int(os.environ.get("MAX_CLIENTS"))
18
      if os.environ.get("MAX CLIENTS") is not None
19
      else float("inf")
20
21)
22 # auth
23 ORG NAME = os.environ.get("ORG NAME")
24 COMMON NAME = os.environ.get("COMMON NAME")
25 # utils
26 MAX_FRAGMENT_SIZE = int(os.environ.get("MAX_FRAGMENT_SIZE"))
```

**5.4.4.2 Logging** A logging.Logger is used to 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.

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

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

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

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

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

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

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

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

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

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

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

```
127.0.0.1 - [10/May/2024 08:38:27] "GET /auth/token HTTP/1.1" 401 -
127.0.0.1 - [10/May/2024 08:38:34] "GET /auth/token HTTP/1.1" 401 -
127.0.0.1 - [10/May/2024 08:38:33] "GET /auth/token HTTP/1.1" 200 -
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/token 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 08:39:13] "GET /friends/ 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 08:39:58] "GET /foinds/ HTTP/1.1" 200 -
127.0.0.1 - [10/May/2024 08:39:58] "GET /lobby/find HTTP/1.1" 201 -
127.0.0.1 - [10/May/2024 08:39:58] "DOST /lobby/create HTTP/1.1" 201 -
127.0.0.1 - [10/May/2024 08:39:58] "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 - # 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:ACK 1:0 00000000 b'\x04;\xee\xd0.\xb1\xb2\xe1\xddw\xb4'...'z' [46:32]>
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 - > ('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:ACK 1:0 00000000 b'\x04;\xee\xd0.\xb1\xb2\xe1\xddw\xb4'...'z' [46:32]>
5001:Inbound - > ('127.0.0.1', 2025) : <0
```

Figure 3: example of server console output

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

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

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

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.

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

### 5.4.4.3.2 Game Model

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

Lobbies are full.

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

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

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

```
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
```

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

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

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

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

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

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

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

```
return False

full return False

full return false

full return (self) -> int:

full return (datetime.datetime.now() - self.heartbeat).seconds

full return false

full return fals
```

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().

```
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 <code>@auth.login\_required</code> decorator). JSON Web Tokens (JWT) are used for session tokens, allowing a user to instead request and use a token for the rest of the session (or until the token expires) instead of using a username and password. This can help mitigate against any man-in-the-middle attacks as, if a token is successfully intercepted, it will only be useable for a limited time and the accounts credentials are not exposed.

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

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

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).

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

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

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

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

### 5.4.4.4.2 Endpoints

The endpoints are implemented according to the API specification.

#### 5.4.4.2.1 Auth

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

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

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

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

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

#### 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.

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

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

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

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

```
else:
abort(404) # no such friends
```

### 5.4.4.2.3 Games

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

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

## 5.4.4.4.2.4 Lobby

The getLobby method is exposed at /lobby and 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.

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

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.

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

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

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

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

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

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

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

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

The getFriendLobbies method at /lobby/friends returns all 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.

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

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

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

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

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

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.

```
1 def validateCertificate(certificate: x509.Certificate, publicKey:
      rsa.RSAPublicKey) -> bool:
       # period
 2
3
      now = datetime.datetime.now(datetime.timezone.utc)
       if not (certificate.not valid before utc <= now <=</pre>
 4
          certificate.not valid after utc): # check in period
           return False
5
 6
       # signature
 7
      try:
           publicKey.verify( # check against publicKey
 8
9
               certificate.signature,
               certificate.tbs certificate bytes,
10
               aPadding.PKCS1v15(),
11
               certificate.signature_hash_algorithm,
12
           )
13
14
       except InvalidSignature:
           return False
15
      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.

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

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

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

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

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

The choice and outcomes are defined in the package <code>\_\_init\_\_</code> allowing both the <code>Client</code> and <code>Server</code> to import them.

```
1 class Choice:
```

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

The Game attributes are defined in game\_config.yaml.

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

These can then be loaded using the yaml package in the \_\_init\_\_.

```
# 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"]
NIN_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.

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

```
if self.onReceiveData:
    self.onReceiveData(addr, data)

10

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

14

15 @staticmethod
16 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.

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

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

The Client utilizes the onConnect to start the gameThread.

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

### 5.4.4.6.2 Server

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

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

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

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

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

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

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

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

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

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

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

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

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

### 5.4.4.7.1 Authentication

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

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

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

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

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.

    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]:
Waiting for other player...
```

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

## 5.4.4.7.3 Matchmaking

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

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

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

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

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

### 5.5 Tests

The pytest module was used to define serval test.

```
platform win32 -- Python 3.11.4, pytest-8.1.1, pluggy-1.5.0 -- C:\Users\harry\bocuments\GitHub\PaperClip\env\Scripts\python.exe
cachedir: _pytest_cache
rootdir: _G:\Users\harry\bocuments\GitHub\PaperClip
collected 16 items

test_udp.py::testItenorCode PASSED

test_udp.py::testItenorCode PASSED

test_udp.py::testItenorPacket PASSED

test_udp.py::testPartPacket PASSED

test_udp.py::testDartAcompress PASSED

test_udp.py::testTacompress PASSED
```

Figure 6: example output of pytest -v

### 6 Conclusion

# 6.1 Project Objective

The main objectives for this project where:

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

Both objectives have been met.

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

The client package provides a user-friendly interface for interactions with the Server. Additionally, using the information provided by the Server, is able to 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 humanreadable 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 (FIF0) 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/201808230150 49/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/l salzman/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 WhitehornProprietary 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.
- This project shall manage storing and retrieving non-volatile information from a database sever. This will include user information and, therefore, handling account validation as well as leaderboards.
- 3. Create a variety of small and simple games to implement the networking and account features.

### Further objectives include:

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

#### Beneficiaries The primary beneficiaries include:

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

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

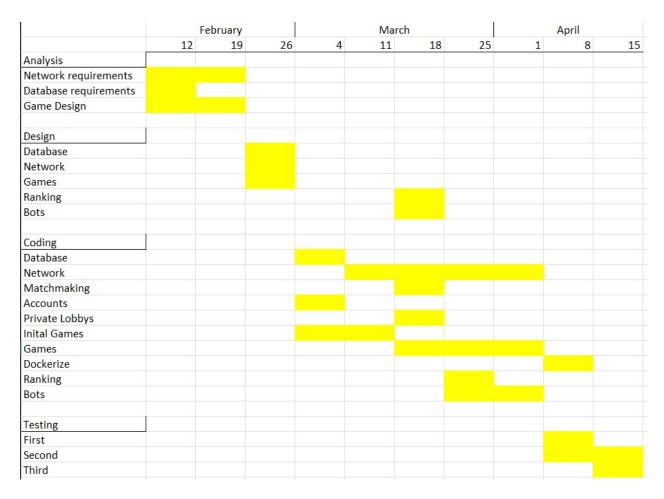


Figure 9: workplan

# Risks

$Objectiv \hbox{\it Risk}$	Severi	tyScore	Risks	Actions
Network2	5	10	Vulnerabilities	Ensure that the system is not vulnerable to any malicious interaction. Including 'cheating'.
Accounts	4	4	GDPR	Limit type of stored data and where data is relevant unsure stored in accordance to GDPR.
Safeguar2ling	4	8	User safety	Limit interactions between user accounts.
Development	5	25	Time	Ensure that progress is made in accordance with work plan and actively update when system requirements change.

ObjectivRisk	Sev	erit\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.)
Matchm3kin	g 1	3	Developing appropriate ELO/rank system may prove to be too complex.	ELO/rank system can be dropped in favour of random matchmaking if the former cannot be developed within the time constraints.
Bots 4	2	8	Developing appropriate bot system may prove to be too complex.	Bot system can be dropped if it cannot be finished in time.

Research Ethics Checklist

See Ethics\_Review\_Form.doc

#### References

- 1. MicroProse, 1996, Civilization II, MicroProse
- 2. DASHBot, (2010). *CivII 01.png*. [image online] Wikipedia. Available at: https://en.wikipedia.org/wiki/File:CivII\_01.png [Accessed 10 Feb. 2024].
- 3. Backbone Entertainment, 2008, Super Street Fighter II Turbo HD Remix, Capcom
- 4. Gamescore Blog, (2008). Super Street Fighter II Turbo HD Remix 12. [image online] Available at: https://www.flickr.com/photos/gamerscore/3058520175 [Accessed 10 Feb. 2024].

## 9.2 Deployment Guide

Requirements Requires: python 3.11+, mysql server.

Note This projects makes use of ANSI color prints.

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

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

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

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

## Python

Setup Optional: python -m venv env and activate
Install packages: pip install -r requirements.txt
Env

Example .env, must be place in app root

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

```
16 ORG_NAME = Paperclip
17 COMMON_NAME = 127.0.0.1
18 ## utils
19 MAX FRAGMENT SIZE = 988
20
21 # client
22 \text{ TCP PORT} = 5000
23
24 # app
25 \; \mathsf{FLASK\_APP} \texttt{=} \mathtt{server}
26 \text{ PRUNE TIME} = 58
27 SECRET_KEY = MyVerySecretKey
28 SQLALCHEMY_DATABASE_URI = mysql://root:root@localhost:3306/paperclip
29
30 # debug
31 DEBUG = True
```

Run Server: python -m flask run

Client: python -m client or python -m client offset (where offset is some int such that C\_PORT (from .env) becomes C\_PORT+=offset)

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

### 9.3 Package

### 9.3.1 udp

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

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

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

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

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

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

```
33
34 def getRsaPrivateFromDer(data: bytes, password: bytes) -> rsa.RSAPrivateKey:
      key = serialization.load der private key(data, password=password)
35
      return key
36
37
38
39 def getDerFromRsaPublic(key: rsa.RSAPublicKey) -> bytes:
      der = key.public bytes(
40
           encoding=serialization.Encoding.DER,
41
           format=serialization.PublicFormat.SubjectPublicKeyInfo,
42
43
      return der
44
45
46
47 def getRsaPublicFromDer(data: bytes) -> rsa.RSAPublicKey:
48
      key = serialization.load_der_public_key(data)
49
      return key
50
51
52 def generateUserCertificate(
      key, userId: int | str | None = None, username: str | None = None
53
54 ) -> x509.Certificate:
      name = \Gamma
55
           x509.NameAttribute(NameOID.ORGANIZATION_NAME, ORG_NAME),
56
           x509.NameAttribute(NameOID.COMMON NAME, COMMON NAME),
57
58
      if userId is not None:
59
           name.append(x509.NameAttribute(NameOID.USER ID, str(userId)))
60
61
      if username is not None:
           name.append(x509.NameAttribute(NameOID.PSEUDONYM, username))
62
      subject = issuer = x509.Name(name)
63
      cert = (
64
           x509.CertificateBuilder()
65
           .subject name(subject)
66
           .issuer name(issuer)
67
           .public key(key.public key())
68
           .serial number(x509.random serial number())
69
           .not valid before(datetime.datetime.now(datetime.timezone.utc))
70
           .not_valid after(
71
               datetime.datetime.now(datetime.timezone.utc) +
72
                  datetime.timedelta(days=1)
73
74
           .add extension(
               x509.SubjectAlternativeName([x509.DNSName("localhost")]),
75
               critical=False,
76
```

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

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

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

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

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

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

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

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

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

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

```
13
      pass
14
15
16 class PaperClipError(Exception):
       """Unknown error"""
17
18
19
20 # connection
21 class ConnectionErrorCodes(Minor):
      CONNECTION = O
22
23
      NO SPACE = 1
      CERTIFICATE INVALID = 2
24
      FINISH_INVALID = 3
25
26
27
28 class ConnectionError(PaperClipError):
       """Handshake connection could not be finished"""
29
30
31
32 class NoSpaceError(ConnectionError):
       """Server has insufficient space to accept new clients"""
33
34
35
36 class CertificateInvalidError(ConnectionError):
       """Certificate is invalid / can not be validated"""
37
38
39
40 class FinishInvalidError(ConnectionError):
41
       """Finish is invalid"""
42
43
44 connectionErrors = {
      ConnectionErrorCodes.CONNECTION: ConnectionError,
45
      ConnectionErrorCodes.NO_SPACE: NoSpaceError,
46
      ConnectionErrorCodes.CERTIFICATE_INVALID: CertificateInvalidError,
47
      ConnectionErrorCodes.FINISH INVALID: FinishInvalidError,
48
49 }
50
51
52 def getConnectionError(minor: ConnectionErrorCodes | int) -> ConnectionError:
53
      try:
           minor = minor if isinstance(minor, Minor) else
54
              ConnectionErrorCodes(minor)
           if minor in connectionErrors:
55
               return connectionErrors[minor]
56
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
1 # udp.packet
 2 import struct
3 from enum import Enum
5 from cryptography.hazmat.primitives.asymmetric.ec import
      EllipticCurvePublicKey
6 from cryptography.x509 import Certificate
8 from . import auth, error, logger, utils
10 \text{ VERSION} = 0
11 # SIZE in Bits
12 VERSION SIZE = 4
13 PACKET TYPE SIZE = 4
14 FLAGS_SIZE = 8
15 SEQUENCE ID SIZE = 16
16 FRAGMENT ID SIZE = 8
17 FRAGMENT_NUM_SIZE = 8
18 INIT VECTOR SIZE = 16
19 CHECKSUM SIZE = 16
20 ACK ID SIZE = SEQUENCE ID SIZE # 16
21 ACK_BITS_SIZE = SEQUENCE_ID_SIZE # 16
22
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
else b""
728
            )
729
730
            certificate = (
                AuthPacket.encodeCertificate(certificate)
731
                if certificate is not None
732
                else b""
733
            )
734
735
            return header + public_key_size + public_key + certificate_size +
               certificate
736
       @staticmethod
737
       def decodeHeader(
738
739
            header: bytes,
       ) -> tuple[
740
            int,
741
742
            Type,
            list[int],
743
744
            int,
            int | None,
745
            int | None,
746
            int | None,
747
            int | None,
748
749
            int,
            EllipticCurvePublicKey,
750
            int | None,
751
            Certificate | None,
752
753
            int.
       ]:
754
755
            *h, offset = Packet.decodeHeader(header)
            public key size = AuthPacket.decodePublicKeySize(header[offset :
756
               offset + 1])
            offset += 1
757
            public key = AuthPacket.decodePublicKey(
758
                header[offset : offset + public key size]
759
760
761
            offset += public key size
            if offset < len(header): # check if more bytes left to decode</pre>
762
                certificate size = AuthPacket.decodeCertificateSize(
763
                    header[offset : offset + 2]
764
765
                )
                offset += 2
766
767
                certificate = AuthPacket.decodeCertificate(
768
                    header[offset : offset + certificate_size]
                )
769
770
                offset += certificate size
```

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

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

```
860
            heartbeat = HeartbeatPacket.encodeHeartbeat(heartbeat)
861
            return header + heartbeat
862
863
864
        @staticmethod
865
        def decodeHeader(
            header: bytes,
866
867
        ) -> tuple[
868
            int.
869
            Type,
            list[int],
870
871
            int,
872
            int | None,
            int | None,
873
874
            int | None,
875
            int | None,
876
            bool,
877
            int,
       ]:
878
            *h, offset = Packet.decodeHeader(header)
879
880
            heartbeat = HeartbeatPacket.decodeHeartbeat(header[offset : offset +
                11)
            offset += 1
881
882
            return *h, heartbeat, offset
883
        # pack / unpack
884
        @classmethod
885
        def _packHeader(cls, p) -> bytes:
886
887
            header = cls.encodeHeader(
                p.version,
888
                p.packet_type,
889
890
                p.flags,
891
                p.sequence id,
                p.fragment id,
892
893
                p.fragment_number,
894
                p.init_vector,
                p.checksum,
895
896
                p.heartbeat,
897
898
            return header
899
900
901 class ErrorPacket(Packet):
902
        _major: error.Major
       minor: error.Minor
903
```

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

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

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

```
11)
             offset += 1
1036
1037
             return *h, major, minor, offset
1038
1039
         # pack / unpack
        @classmethod
1040
        def packHeader(cls, p) -> bytes:
1041
1042
             header = cls.encodeHeader(
1043
                 p.version,
                 p.packet type,
1044
                 p.flags,
1045
                 p.sequence_id,
1046
                 p.fragment_id,
1047
                 p.fragment number,
1048
                 p.init vector,
1049
1050
                 p.checksum,
1051
                 p.major,
1052
                 p.minor,
1053
             )
             return header
1054
1055
1056
1057 def unpack(rawP:bytes) -> Packet:
        packet_type = Packet.decodeVersionType(rawP[0:1])[1]
1058
        match packet_type:
1059
             case Type.DEFAULT:
1060
                 return Packet.unpack(rawP)
1061
             case Type.ACK:
1062
1063
                 return AckPacket.unpack(rawP)
1064
             case Type.AUTH:
                 return AuthPacket.unpack(rawP)
1065
             case Type.HEARTBEAT:
1066
                 return HeartbeatPacket.unpack(rawP)
1067
             case Type.ERROR:
1068
                 return ErrorPacket.unpack(rawP)
1069
1070
             case :
                 logger.warning(f"Cannot unpack '{packet type}' due to invalid
1071
                     packet type.")
```

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

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

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

```
) -> tuple[packet.AuthPacket, tuple[str, int]]:
90
            if addr not in self.clients: # new client
91
92
                if self.isNotFull(): # check space
                    # print(f"{bcolors.WARNING}# Handshake with {addr}
93
                        starting.{bcolors.ENDC}")
                    logger.info(
94
                        f"{bcolors.WARNING}# Handshake with {addr}
95
                            starting. {bcolors.ENDC}"
96
                    valid, accountId = self.validateCertificate(p.certificate)
97
                    if not valid:
98
                        # raise ValueError(f"Invalid peer cert {p.certificate}")
99
                        logger.error(f"Invalid peer cert {p.certificate}")
100
                        self.queueError(
101
102
                            addr,
103
                            major=error.Major.CONNECTION,
                            minor=error.ConnectionErrorCodes.CERTIFICATE INVALID,
104
                            data=b"Invalid Certificate.",
105
                        )
106
                    else:
107
                        self.makeClient(addr, p.certificate, accountId)
108
                        self.regenerateEcKey(addr)
109
                        sessionKey = auth.generateSessionKey(
110
                            self.getEcKey(addr), p.public key
111
                        )
112
                        self.setSessionKey(addr, sessionKey)
113
                        self.queueAuth(addr, self.cert,
114
                            self.getEcKey(addr).public key())
115
                        self.queueFinished(addr, p.sequence_id,
                            self.getSessionKey(addr))
                else:
116
                    # print(f"{bcolors.FAIL}# Handshake with {addr} denied due to
117
                       NO SPACE. {bcolors.ENDC}")
                    logger.warning(
118
                        f"{bcolors.FAIL}# Handshake with {addr} denied due to
119
                           NO SPACE. {bcolors.ENDC}"
120
121
                    self.queueError(
122
                        addr,
123
                        major=error.Major.CONNECTION,
                        minor=error.ConnectionErrorCodes.NO SPACE,
124
125
                        data=b"Server is Full.",
                    )
126
            else:
127
                sessionKey = auth.generateSessionKey(self.getEcKey(addr),
128
```

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

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

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

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

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

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

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

```
1 # udp.utils
2 import zlib
3
```

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

## 9.3.2 server

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

```
14 PRUNE TIME = int(os.environ.get("PRUNE TIME"))
15
16
17 def create app():
      app = Flask(__name__)
18
19
      app.jinja env.trim blocks = True
20
      app.jinja_env.lstrip_blocks = True
21
22
      app.config["SECRET KEY"] = os.environ.get("SECRET KEY").encode()
23
      uri = os.environ.get("SQLALCHEMY DATABASE URI")
24
      init = False
25
      if not database_exists(uri):
26
          init = True
27
28
          create database(uri)
29
      app.config["SQLALCHEMY_DATABASE_URI"] = uri
30
      db.init_app(app) # noqa: F405
31
32
33
      with app.app context():
          db.create_all() # noqa: F405
34
35
36
      if init:
          with app.app_context():
37
38
               # init games
              from rps import ID, NAME, MIN PLAYERS, MAX PLAYERS
39
              Statement.createGame(ID, NAME, MIN PLAYERS, MAX PLAYERS) # noqa:
40
                  F405
41
               # example accounts
              m = Statement.createAccount("Mario", "ItsAMe123") # noga: F405
42
              p = Statement.createAccount("Peach", "MammaMia!") # noga: F405
43
              b = Statement.createAccount("Bowser", "M4r10SucK5") # noqa: F405
44
              Statement.createFriends(m.id, p.id) # noqa: F405
45
              Statement.createFriends(p.id, b.id) # noqa: F405
46
47
48
      from .main import main as main_blueprint
49
      app.register blueprint(main blueprint)
50
51
52
      return app
```

```
1 # server.lobbies
2 import os
```

```
4 import dotenv
5 from flask import Flask
7 from udp import logger # noqa: F401
9 from .models import * # noqa: F403
10
11 from sqlalchemy utils import database exists, create database
12
13 dotenv.load dotenv()
14 PRUNE TIME = int(os.environ.get("PRUNE TIME"))
15
16
17 def create app():
      app = Flask(__name__)
18
19
      app.jinja env.trim blocks = True
20
      app.jinja env.lstrip blocks = True
21
22
      app.config["SECRET_KEY"] = os.environ.get("SECRET_KEY").encode()
23
      uri = os.environ.get("SQLALCHEMY DATABASE URI")
24
25
      init = False
      if not database_exists(uri):
26
27
          init = True
          create database(uri)
28
      app.config["SQLALCHEMY DATABASE URI"] = uri
29
30
31
      db.init_app(app) # noqa: F405
32
      with app.app context():
33
          db.create_all() # noqa: F405
34
35
      if init:
36
37
          with app.app_context():
               # init games
38
              from rps import ID, NAME, MIN PLAYERS, MAX PLAYERS
39
              Statement.createGame(ID, NAME, MIN PLAYERS, MAX PLAYERS) # noqa:
40
                  F405
               # example accounts
41
              m = Statement.createAccount("Mario", "ItsAMe123")
                                                                   # noga: F405
42
              p = Statement.createAccount("Peach", "MammaMia!") # noqa: F405
43
              b = Statement.createAccount("Bowser", "M4r10SucK5") # noqa: F405
44
              Statement.createFriends(m.id, p.id) # noqa: F405
45
              Statement.createFriends(p.id, b.id) # noqa: F405
46
```

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

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

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

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

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

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

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

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

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

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

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

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

## 9.3.3 rps

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

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

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

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

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

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

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

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

```
150
                                     o = f"You {bcolors.OKCYAN}DRAW{bcolors.ENDC}.
                                 case _:
151
                                    0 = ""
152
153
                            print(
                                 f"\n{o}You Picked {data['choice']}. They picked
154
                                    {data['otherChoice']}.\nThe score is
                                    {data['score']['score']}:{data['otherScore']['score']}."
155
                            if data["outcome"] == Outcome.WIN:
156
                                 self.score += 1
157
                            self.recvQueue.task_done()
158
           finally:
159
160
                self.udpClient._quit()
161
162
       def quit(self, msg: str = "quit call", e: Exception | None = None) ->
           None:
            self.isRunning = False
163
            self.udpClient.quit(msg, e)
164
```

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

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

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

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

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

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

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

## 9.3.4 client

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

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

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

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

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

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

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

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

```
248 def _friends(client: Client):
       while True:
249
            print(f"{bcolors.HEADER}\nFriends.{bcolors.ENDC}")
250
            friends = client.getFriends()
251
252
            friends = "\n\t".join(
253
                Γ
                    f"{i+1}. {friend['username']}"
254
255
                    for i, friend in enumerate(friends["friends"])
                ٦
256
257
            )
            print(f"Friend list: \n\t{friends}")
258
            print("\n1. Add New Friend\n2. Remove Friend\n3. Return to Main Menu")
259
260
            while True:
                option = input(": ").strip()
261
262
                match option:
                    case "1" | "2":
263
264
                        username = input("\nUsername: ").strip()
265
                        match option:
                             case "1":
266
267
                                 try:
268
                                     client.addFriend(username)
269
                                     print(
                                          f"\n{bcolors.OKGREEN}Account '{username}'
270
                                             added as friend{bcolors.ENDC}"
271
272
                                     break
273
                                 except AssertionError:
274
                                     print(
275
                                          f"\n{bcolors.FAIL}Error: No such account
                                             with username
                                             '{username}'.{bcolors.ENDC}"
276
                                     )
                             case "2":
277
278
                                 try:
279
                                     client.removeFriend(username)
280
                                     print(
                                          f"\n{bcolors.OKGREEN}Account '{username}'
281
                                             removed as friend{bcolors.ENDC}"
282
                                     break
283
284
                                 except AssertionError:
285
                                     print(
286
                                          f"\n{bcolors.FAIL}Error: No such account
                                             with username '{username}' in friend
                                             list.{bcolors.ENDC}"
```

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

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

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

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

```
1 # .env
2 # udp
3 S_HOST=127.0.0.1
4 S PORT=2024
5 C HOST=127.0.0.1
6 C_PORT=2025
7 ## node
8 SOCKET BUFFER SIZE = 1024
9 SEND SLEEP TIME = 0.1
10 QUEUE TIMEOUT = 10
11 SOCKET TIMEOUT = 20
12 ## server
13 HEARTBEAT MAX TIME = 120
14 HEARTBEAT_MIN_TIME = 30
15 MAX CLIENTS
16 ## auth
17 ORG_NAME = Paperclip
18 COMMON NAME = 127.0.0.1
19 ## utils
20 MAX_FRAGMENT_SIZE = 988
21
22 # client
```

```
TCP_PORT = 5000

24

25  # app

26 FLASK_APP = server

27 PRUNE_TIME = 58

28 SECRET_KEY = MyVerySecretKey

29 SQLALCHEMY_DATABASE_URI = mysql://root:root@localhost:3306/paperclip

30

31  # debug

32 DEBUG = True
```

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

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

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

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

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

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

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

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

**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.

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

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

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

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

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

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

## 9.4 Packet Specification

## **Packets**

### **DEFAULT** DEFAULT Packet Specification:

															DE	FAU	JLT	Pack	et														
Offsets															2							3	3										
Octet	Bit	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	0 Version Type Flags									Sequence ID																						
4	32	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	515

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	RF	ESE	RVED

## 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	Pa	cket															
Offsets	Octet				0								1						2	2							- :	3			
Octet	Bit	0	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												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	$ \begin{array}{ c c c c c c c c c c c c c c c c c c c$											
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.

TITTA	DT	DI	٦ ٨ ٦	$\Gamma$	1 - 4 -
HEA	R.L	D	$_{I}A$	1 1 2	)ata

Unused field.

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

## ERROR packet specification:

															E	RRC	R P	acke	t														
Offsets	Octet				0									1							2	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	4 Default Packet Headers											F	Error Major Error Minor Date					$ta^*$														

## 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

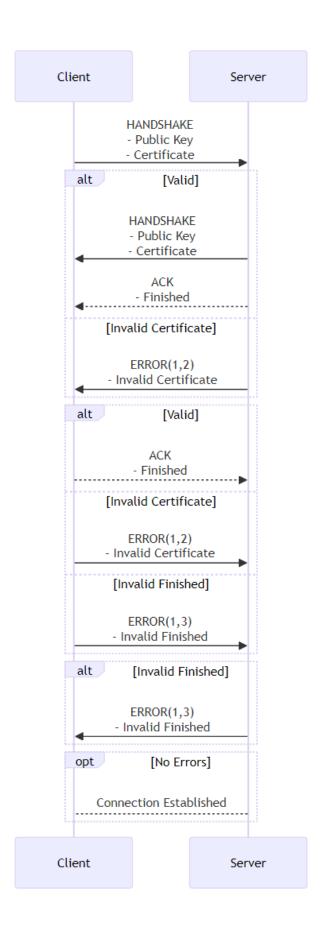
This field allows for additional information about the error to be shared. This used for logging purposes but can also be used for displaying *user-friendly* error messages to a client.

## **Error Codes** The error codes are defined as follows:

		Er	rror Codes						
Major	Minor	Name	Description						
	0	CONNECTION	Connection Handshake Could Not Finish						
1	1	NO SPACE	Server has no more space						
т	2	CERTIFICATE INVALID	Certificate is invalid / cannot be validated						
	3	FINISH INVALID	Finished is invalid						
	0	DISCONNECT	A Party is Disconnecting						
<b>2</b>	1	SERVER DISCONNECT	The server is closing, all clients must exit gracefully						
	2	CLIENT DISCONNECT	The client is closing, the server must handle gracefully						
	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						
3	4	SEQUENCE ID	Sequence id does not match expected						
J	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						
415		RESERVED							

## **Behavior**

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



## Session Key

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

#### Finished

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

## Certificate Validation

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

## Flags Behaviour Reliable

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

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

#### Checksum

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

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

### Compressed

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

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

### Encrypted

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

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

## Frag

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

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

### **Disconnection** Client Disconnect

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

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

#### Server Disconnect

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

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

## 9.5 API Specification

Login	Name	Endpoint	Method	Inputs	Returns	Description						
				A	uth							
	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.						
				G	ames							
	Get Games	/games	GET		list[Game]	Return a list of all available games.						
				L	obby							
	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						
				Fr	iends							
	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			Username		Remove an Account as a Friend						

# 9.6 ERD Diagram

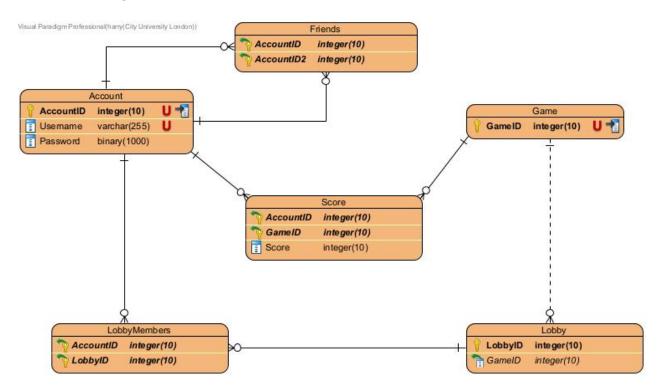


Figure 10: Database Models ERD