POLISH-JAPANESE ACADEMY OF INFORMATION TECHNOLOGY

The Faculty of Information Technology

Department of Software Engineering

Software, business process and database engineering

Adam Furche
s29706

High-level engine for networking of online multiplayer games in Python

MSc thesis written under the supervision of:

Mariusz Trzaska Ph.D.

Warsaw, January, 2025

Summary

Online video games are rapidly rising in popularity. Their development requires extensive computer networking knowledge and experience handling problems unique to distributed systems. This results in more tools being released to support their creation in many different technologies, including Python programming language. Many libraries attempt to relieve developers from those pains by abstracting complexity while offering a simple interface. However, an analysis of existing solutions for Python pointed out their limitations and weaknesses. The thesis presents the concept of a competitive high-level engine designed with this particular programming language in mind. The work contains a discussion of the challenges such a solution has to address and a list of required functionalities formulated based on research and practical experiences. Additionally, the performance limitations of Python were investigated. For this purpose, multiprocessing and asynchronous programming were tested to distinguish an alternative for multithreading limited by the *Global Interpreter Lock*. A culmination of the work consists of a presentation of an implemented prototype of a high-level engine, showcasing its core functionalities and their use cases.

Keywords: online multiplayer games, software engine, concurrency in Python programming language

POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Adam Furche
Nr albumu s29706

Wysokopoziomowy silnik do obsługi funkcjonalności sieciowych gier wieloosobowych w języku Python

Praca magisterska napisana pod kierunkiem:

Dr inż. Mariusz Trzaska

Warszawa, styczeń, 2025

Streszczenie

W dzisiejszych czasach gry sieciowe coraz bardziej zyskują na popularności. Ich rozwój wymaga obszernej wiedzy z dziedziny sieci komputerowych oraz doświadczenia w rozwiązywaniu problemów unikalnych dla systemów rozproszonych. Z tego powodu powstaje coraz więcej narzędzi wspierających ich tworzenie w różnych technologiach, w tym języku Python. Wiele bibliotek próbuje odciążyć programistów od wspomnianych bolączek poprzez abstrahowanie złożoności przy jednoczesnym oferowaniu prostego interfejsu. Jednakże, analiza istniejących rozwiązań dla języka Python wskazuje na ich ograniczenia i słabości. Praca prezentuje koncept konkurencyjnego wysokopoziomowego silnika, zaprojektowanego mając na uwadze ten konkretny język programowania. Przedstawiona została dyskusja na temat wyzwań, które powinny zostać zaadresowane przez rozwiązanie oraz lista wymaganych funkcjonalności, sformułowana na podstawie badań i praktycznych doświadczeń. Dodatkowo przeanalizowane zostały ograniczenia wydajnościowe języka Python. W tym celu przetestowano wieloprocesowość oraz programowanie asynchroniczne, w celu wyłonienia alternatywy dla wielowątkowości ograniczonej przez tzw. *Global Interpreter Lock*. Zwieńczeniem pracy jest zaprezentowanie zaimplementowanego prototypu wysokopoziomowego silnika, rzucające światło na najważniejsze funkcjonalności oraz ich zastosowania.

Słowa kluczowe: wieloosobowe gry sieciowe, silnik, współbieżność w języku Python

Table of contents

1. Introduction

Online multiplayer games have significantly risen in popularity in recent years. There are many reasons why, such as the fact that competition with players from around the world can be very absorbing and they also give the possibility to spend pleasant time cooperating with friends in a virtual world. Due to that interest, the development of such video games also is much more prevalent. However, many programmers, especially beginners might hit a crucial roadblock in their journey. The creation of online multiplayer games requires a substantial amount of technical knowledge in the domain of computer networking and practical experience to deal with challenges unique to distributed systems. With those pains in mind, many frameworks and tools have been developed to lower the entry-level to this subject and enhance the productivity and satisfaction of the developers of multiplayer games.

1.1. Work context

The idea that motivates the work is a product of the author's experiences with developing a few multiplayer games for university and personal projects, and his sympathy for the Python programming language. The existing solutions for this technology were not found to be satisfying enough. On the other hand, working briefly with the *Mirror Networking* framework for the *Unity Game Engine* [1] described in section 2.1 showed that such tools can be significantly more impactful. This posed the question, why cannot there exist a solution for Python that would improve the creation of online multiplayer games to such an extent?

1.2. Goals

The main objective of the work is to define a concept of a high-level engine for the networking of online multiplayer games in Python, that would be competitive with existing solutions. To achieve that firstly all most notable frameworks need to be analysed to discover their strengths and weaknesses. Knowing that the Python language is generally regarded as not performant enough for such complex use cases, it also requires addressing. Therefore another goal of the project is to research possible implementation ways and assess their efficiency to choose one that is most suitable for the task. Lastly, a functional engine prototype is to be developed to prove that the design is feasible to implement and solves the stated problems.

1.3. Expected results

It is expected that the work will produce the following results:

- List of required functionalities that a high-level engine for networking of online multiplayer games should possess. It should encapsulate all the necessary characteristics to provide foundations for the framework design.
- Results of performance tests showing the potential efficiency of simple prototypes based on different concurrency approaches. Such experiments are necessary prerequisites to choose the architecture of the engine properly.
- A working engine prototype of a high-level engine for networking of multiplayer games written in Python. It should implement requirements that were to be concluded based on research of existing solutions and practical experiences.

- At least one introductory sample project showcasing the engine's functionalities. Such is required to provide a convenient entry point for new users and can be used in performance tests.
- Results of performance tests conducted on the engine prototype using one of the introductory sample projects. This can give initial insight into the expected performance of the final solution.

1.4. Document organization

The document is organized into 6 chapters:

- The first chapter provides a brief introduction to the topic, outlining the author's motivation behind the topic idea, goals, and expected results.
- The second chapter contains a detailed analysis and discussion of existing solutions, stating their strengths and weaknesses.
- The third chapter presents the concept of a new solution for networking of online multiplayer games, stating crucial requirements and reasoning behind them. Additionally, it concerns the topic of Python concurrency limitations, showing implemented engine prototypes and results of conducted performance tests
- The fourth chapter briefly describes all technologies, libraries, and tools used throughout the prototype development.
- The fifth chapter explains in detail the architecture of the implemented engine prototype, showcasing the core functionalities and their use cases. Moreover, it also includes results of initial solution performance tests and a discussion about its strengths, areas of possible improvements, and plans for future development.
- The sixth chapter concludes the work, summarizing the document and achieved results.

2. Existing solutions

This chapter gives insight into existing solutions for online multiplayer game networking, showcasing the Mirror library for Unity Game Engine as one of the author's inspiration sources. It also describes Python frameworks currently available on the market, pointing out their strengths and weaknesses.

2.1. Mirror Networking Library

As the development of online multiplayer games requires rather extensive knowledge of computer networking, many frameworks exist to exempt programmers from the need to focus on low-level network programming, providing a more friendly interface on a higher level of abstraction. As mentioned previously, one of the author's biggest inspirations was personal experience working with an open-source high-level library for Unity Game Engine: Mirror. Although now it is known by the aforementioned name, it was originally developed as a first-party Unity framework called UNET. The main idea behind this tool, as presented in a speech given during the Unite Europe 2015 event [2] by members of a team dedicated to the project, Erik Juhl and Larus Olafsson, was to "democratize" multiplayer development. What it meant was to enable all game developers to create multiplayer games they dreamed of making, relieving them from many so-called "multiplayer pains", such as the need for strong domain-specific knowledge, including socket APIs and transport protocols. As described in the development blog titled "A Brief History of Mirror" on the official website of Mirror Networking [1], UNET brought up a few compelling functionalities, such as the possibility to share the majority of the code between the client and server, automate the serialization of chosen variables, and automatically call functions from a client on a server using a wrapper. Such features significantly simplify multiplayer game development, as thanks to them it's possible to implement both client and server in a single project, and all the work required to send and serialize/deserialize messages would be entirely handled for the developer by the framework. Even though UNET was not finalized by the Unity team and was ultimately abandoned, the open-source community took charge of the project, and thanks to the immense work of contributors it came to fruition as the Mirror Networking library. The result turned out to be a success as it gained a major developer audience, having around 100'000 downloads a year [1], one of them being the author of the thesis, as he had an opportunity to use it while implementing a Real-Time Strategy game for his Bachelor's thesis.

The biggest strength of the Mirror library is its ease of use. As its design follows the principles of the Unity Game Engine API, the development of a multiplayer game doesn't differ greatly from creating a single-player one. Video games made in Unity Game Engine consist of a tree of *GameObjects*, which can represent various entities, such as characters and buildings, as well as more abstract beings, such as cameras and lighting [3]. Their functionalities are defined using functional pieces named *Components* that can be assigned to them [4]. Developers can either choose from a variety of different pre-implemented *Components*, or they can define ones themselves using a C# script. Mirror introduces several useful *Components*, the most significant being the *Network Manager*, which encapsulates numerous functionalities for managing networking of multiplayer games, such as shared game state and object spawn management [5]. Additionally, the library introduces the *Network Identity Component*, which assigns objects a unique identifier used in the networking system. It's used by the framework in its one of the most crucial features, as objects using it that are instantiated on the game server, will be automatically instantiated on all of the connected clients as well, relieving the developer of the need to implement it by themselves [6]. Often the prebuilt *Components* are not sufficient enough to accomplish all the required logic, hence it is necessary to create custom scripts. A

simple example of such a code snippet can be seen in listing 2.1, which presents an exemplary script from the author's Bachelor thesis project [7].

Listing 2.1 Example of a *Unity Game Engine* script Component (source: [7])

```
public class MusicManager : MonoBehaviour {
    [SerializeField] AudioSource backgroundMusic;
    [SerializeField] bool isMainMenu = false;
    void Start() {
        backgroundMusic.volume = 0.1f;
        if (isMainMenu) backgroundMusic.time = 2f;
    }

    public void ChangeVolume(float newVolume) {
        backgroundMusic.volume = newVolume;
    }
}
```

The functionalities can be added by creating a class for a specific *GameObject* that inherits from the `MonoBehaviour` class. This allows developers to implement callbacks executed by the *Unity Game Engine* on particular events, e.g. the `Start()` which will be run after an object instance is created, and the `Update()` which will be called on each new frame update [8]. As depicted in listing 2.2, Mirror Networking follows this design principle introducing the `NetworkBehaviour` base class, to be used with *GameObjects* containing the *Network Identity Component*. On top of the functionalities of the `MonoBehaviour`, it comes with new special hooks, such as the `OnStartServer()` that is executed when an object is spawned on the server, and the `OnStartClient()` running after instantiation on the client side [9]. Additionally, the Network Behaviour Component contains two vital features, *SyncVars* and *Commands/ClientRPCs*. *SyncVars* are wrappers to supported data types, that automatically synchronize object properties from the server to all clients. What it means is that where a certain variable value is modified on the game server, the update will be propagated to all connected clients, without any need for manual implementation [10]. On the other hand *Commands* and *Client RPCs* introduce a way to call a defined class method on the server triggered and parameterized on the client and vice versa. The entire process of serializing function parameters, sending a message, receiving, deserializing it on the other side, and executing the method is done by the framework, which significantly reduces the work required to be done by the developer, additionally making it possible to share both client and server functionalities to be implemented in a single class, in contrast to having to create to separate ones [11], [12].

Based on the described features of the Mirror Networking library, one can see the philosophy behind the framework, which focuses strongly on ease of use and programmers' satisfaction. It tries to reduce the high entry threshold into the development of multiplayer games and the amount of work required to achieve the necessary functionalities.

Listing 2.2 Example of a *Unity Game Engine* script *Component* using the `NetworkBehaviour` base class from the *Mirror Networking Library* (source: [7]).

```
public class HealthIndicator : NetworkBehaviour {
    [SerializeField] private HealthSystem healthSystem;
    private Transform healthBar;

    public override void OnStartClient() {
        base.OnStartClient();
        healthBar = transform.Find("HealthPrefab/healthBar");
        healthBar.localScale = new Vector3(1,1,1);
    }

    public override void OnStartServer() {
```

```
            base.OnStartServer();
            healthSystem.OnDamaged += HealthManager_OnDamaged;
        }

    [Server]
    private void HealthManager_OnDamaged(object sender, Vector3 sliderScale) {
        RpcModifyHealthBar(sliderScale);
    }

    [ClientRpc]
    private void RpcModifyHealthBar(Vector3 sliderScale){
        healthBar.localScale = sliderScale;
    }
}
```

2.2. Existing Python Frameworks

Even though Python game development is a niche in comparison to Unity Game Engine, there have already been many attempts to create a framework that takes care of the networking functionalities of online multiplayer games. This section of the thesis will briefly describe the most notable solutions currently available on the market, pointing out their strengths and weaknesses.

2.2.1 PodSixNet

PodSixNet [13] is a library that works by sending events asynchronously and executing assigned custom callbacks (which should be defined with the prefix `Network_*`) when a particular event is received.

Listing 2.3 Example of a client implementation using *PodSixNet* (source: [13])

```
class Client(ConnectionListener):
    def __init__(self, host, port):
        self.Connect((host, port))
        print("Chat client started")
        print("Ctrl-C to exit")
        # get a nickname from the user before starting
        print("Enter your nickname: ")
        connection.Send({"action": "nickname", "nickname":  stdin.readline().rstrip("\n")})
        # launch our threaded input loop
        t = start_new_thread(self.InputLoop, ())

    def Loop(self):
        connection.Pump()
        self.Pump()

    def InputLoop(self):
        # horrid threaded input loop
        # continually reads from stdin and sends whatever is typed to the server
        while 1:
            connection.Send({"action": "message", "message": stdin.readline().rstrip("\n")})

    ######################################
    ### Network event/message callbacks ###
    ######################################

    def Network_players(self, data):
        print("*** players: " + ", ".join([p for p in data['players']]))

    def Network_message(self, data):
        print(data['who'] + ": " + data['message'])
```

```
        # built in stuff

        def Network_connected(self, data):
            print("You are now connected to the server")

        def Network_error(self, data):
            print('error:', data['error'][1])
            connection.Close()

        def Network_disconnected(self, data):
            print('Server disconnected')
            exit()
```

To implement client functionalities, the developer is required to create a class inheriting from a `ConnectionListener` base class, which provides methods for connection and communication with a server, including:

- `ConnectionListener.Connect((<host>,<port>))` - a method that allows to open a connection with a server.
- `ConnectionListener.connection.Send(<event>)` - a method that allows sending a network event, as a Python dictionary. This dictionary should include a field named "*action*", which states which network callback should be executed (the value should be equal to the name of the defined method after the `Network_*` prefix), and other fields of any type.
- `ConnectionListener.connection.Pump()` - a method that calls all callbacks based on received network events. It should be executed once per game loop iteration.

An implementation following the above-mentioned design can be found in listing 2.3, taken from an example project providing simple chatting functionality, available in the official GitHub repository of *PodSixNet*. It depicts how the above-mentioned methods could be used, and also shows a template on how network event callbacks ought to be defined.

Listing 2.4 Example of a server implementation using *PodSixNet* (source: [13])

```
    class ClientChannel(Channel):
        """
        This is the server representation of a single connected client.
        """
        def __init__(self, *args, **kwargs):
            self.nickname = "anonymous"
            Channel.__init__(self, *args, **kwargs)

        def Close(self):
            self._server.DelPlayer(self)

         ##################################
         ### Network specific callbacks ###
         ##################################

        def Network_message(self, data):
            self._server.SendToAll({"action": "message",
                                    "message": data['message'],
                                    "who": self.nickname})

        def Network_nickname(self, data):
            self.nickname = data['nickname']
            self._server.SendPlayers()

    class ChatServer(Server):
        channelClass = ClientChannel

        def __init__(self, *args, **kwargs):
```

```
            Server.__init__(self, *args, **kwargs)
            self.players = WeakKeyDictionary()
            print('Server launched')

        def Connected(self, channel, addr):
            self.AddPlayer(channel)

        def AddPlayer(self, player):
            print("New Player" + str(player.addr))
            self.players[player] = True
            self.SendPlayers()
            print("players", [p for p in self.players])

        def DelPlayer(self, player):
            print("Deleting Player" + str(player.addr))
            del self.players[player]
            self.SendPlayers()

        def SendPlayers(self):
            self.SendToAll({"action": "players", "players": [p.nickname for p in self.players]})

        def SendToAll(self, data):
            [p.Send(data) for p in self.players]

        def Launch(self):
            while True:
                self.Pump()
                sleep(0.0001)
```

Exemplary server implementation can be found on the listing 2.4. The programmer is required to create two subclasses:

- A subtype of the `Channel` base class, which would represent a single connected client. This subclass should include the implementation of callbacks for specific network events.
- A subtype of the `Server` base class, which would work as the game server. In it, it is necessary to configure the subtype of the *Channel* class that will be used for handling client connections using a class property `channelClass`. It will be instantiated when executing a `Connected(<channel>, <addr>)` method that also needs to be defined by the developer. Inside this implementation, it is required to save a reference to the created channel, as presented in the `AddPlayer(self, player)` method in the listing 2.4. This is necessary as an instance of this object is required for sending messages to a specific client using a `Channel.Send(<event>)` method.

Strengths of *PodSixNet*:

- It is lightweight and simple to use.
- It is open-source and free to use in commercial projects (LGPL-3.0 license).
- It is written in pure Python, without external libraries, which guarantees portability.
- The performance should be sufficient due to the usage of asynchronous programming (*asyncore* library) [13].
- It has extensive documentation provided in the form of 3 usage examples and README on the library's official GitHub repository.
- It is publicly available on PyPI.

Weaknesses of *PodSixNet*:

- The drawback of the simplicity of the library is that various crucial aspects of an online multiplayer game, such as the sharing of game state between the server and all client instances,

movement of objects, synchronization of object properties, and game lobby require custom implementation.

- It is not actively supported anymore and uses deprecated libraries. As mentioned in the official Python documentation [14] the crucial *asyncore* library has been deprecated since Python 3.6 and was ultimately removed in version 3.12.

2.2.2 PyGaSe

PyGaSe [15] is a library that provides an implementation of game state (`GameState` class), which can be initialized with custom attributes and is automatically synchronized between the server and all connected clients. Additionally, it gives the possibility to send events and execute custom procedures when a specific event is received.

On the client side, the programmers can create instances of a `Client` class, which provides functions for connecting to and communicating with a server, including:

- `Client.connect_in_thread(<hostname>, <port>)` - a method used for establishing the connection with a server.
- `Client.register_event_handler(<event_type>, <event_handler_function>)` - a method used to declare a method (`event_handler_function`), that will be called when an event of a specific type (`event_type`) is received.
- `Client.dispatch_event(<event_type>, <ack_handler>)` - a method that sends an event to a server, and calls the specified function (`ack_handler`) when an event is received.
- `Client.access_game_state()` - a method that allows for safe access to the synchronized `GameState` by returning a context manager. Context managers are Python types that can be utilized to define a limited runtime context in a program using the *with* clause [17]. The use of the function is presented in listing 2.5, which depicts a connection to a server and access to a game state using the `Client` class in a live Python session.

Listing 2.5 Exemplary usage of *PyGase's* Client class functionalities (source: [16])

```
>>> from pygase import Client
>>> client = Client()
>>> client.connect_in_thread(port=8080)
<Thread(Thread-1, started 24096)>
>>> with client.access_game_state() as game_state:
...     print(game_state.position)
...     print(game_state.hp)
...
0.7904802223420048
100
```

For the development of the server functionalities, the library provides a *Backend* class, exemplary initialization of which can seen in listing 2.6. In its constructor, it is required to pass an instance of the `GameState` class and a definition of the `time_step` function, which represents a single iteration of the game loop. Additionally, the programmer can add a Python dictionary of handler functions for particular events. The game server can be started using a `Backend.run()` method, which starts a new Python thread with a game loop that handles the game logic defined in handlers and synchronization of the `GameState` across the server and all connected clients. It is also possible to send events from a server to a single client, or many clients using the `Backend.server_dispatch_event()` method.

Listing 2.6 Example of server implementation using *PyGase* (source: [16])

```
import math
from pygase import GameState, Backend

# Let there be an imaginary enemy at position 0 with 100 health points.
initial_game_state = GameState(position=0, hp=100)

def time_step(game_state, dt):
    # Make the imaginary enemy move in sinuous lines like a drunkard.
    new_position = game_state.position + math.sin(dt)
    return {"position": new_position}

backend = Backend(initial_game_state, time_step)
backend.run('localhost', 8080)
```

Strengths of *PyGaSe* library:

- High-level, easy-to-use library, with automatized synchronization of a defined game state on the server and all client instances.
- The performance should be sufficient due to the usage of a mix of asynchronous programming and threading.
- Ensures communication reliability, as it handles package losses and network congestions.
- It is open-source and free to use in commercial projects (MIT license).
- Has an extensive documentation in HTML format.
- It is publicly available on PyPI.

Weaknesses of *PyGase* library:

- The game state does not handle player ownership of contained objects, hence it's the programmer's responsibility to properly restrict update possibilities.
- It does not include the game lobby and object movement functionalities.
- The project is not actively supported anymore. The latest version was released in May of 2019.

2.2.3 Simple-Game-Server

Simple-Game-Server [18] is a library that operates by creating many *rooms* on the server side, to which clients can connect.

An example of client implementation is presented in listing 2.7. Programmers are required to create an instance of the `Client` class, which provides methods that can be used to connect and communicate with other clients, as well as to see and join available *rooms*, including:

- `Client.get_rooms()` - a method that returns a list of *rooms* (provides an identifier, number of players, and capacity of all *rooms*).
- `Client.join(<room_id>)` - a method that allows a client to connect to a *room* using its identifier.
- `Client.send(<data>)` - a method that allows to send data to all clients connected to a *room*.
- `Client.sendto(<player_id>, <data>)` – a method that allows to send data to a specified client.
- `Client.get_messages()` - a method that returns a collection of all messages received by a client.

Listing 2.7 Example of client implementation using *Simple-Game-Server* (source: [18])

```
# Add Client instance to your game
client = Client("127.0.0.1", 1234, 1234, 1235)

# Get room list (room_id, nb_players, capacity)
rooms = client.get_rooms()

# You can join a room using room identifier (ex: first room)
client.join(rooms[0]["id"])

# You can autojoin the first available room client.autojoin()
# Or you can create a new room with a client.create_room("room_name")

# In your game main loop
while game_is_running:
    # Data to send
    data = {"foo": "bar", "baz": "gu"}

    # Send data to all players in the room
    client.send(data)

    # Send data to one player in the room
    client.sendto(someone.identifier, data)

    # Send data to multiple players in room
    players_ids = [player1.identifier, player2.identifier]
    client.sendto(players_ids, data)

    # Read received messages
    messages = client.get_messages()
    if len(messages) != 0:
        for message in message:
            do_something_with_message(message)
```

The server's implementation's only purpose is to act as a proxy for the client's communication, therefore it is not possible to add any custom logic. It can be started using the command shown in listing 2.8.

Listing 2.8 The command for running a game server using *Simple-Game-Server* (source: [18])

```
python server.py --tcpport 1234 --udpport 1234 --capacity 10
```

Strengths of Simple-Game-Server:

- It is capable of handling multiple *rooms* at once, which allows it to run multiple game instances concurrently.
- It is an open-source project free to use in commercial projects (GPL-2.0 license).

Weaknesses of Simple-Game-Server:

- It has very limited functionalities, as apart from *room* logic, there is only a possibility to send simple messages.
- It is entirely client-authoritative, which can cause issues with game state synchronization and makes cheating significantly easier.
- The documentation requires improvements as only a short README is provided as documentation and there are no examples of use.
- It is not available in PyPI or other online package indexes.

2.2.4 MpGameServer

MpGameServer [19] is the most powerful framework out of all presented in the thesis, though it comes at a cost of high complexity. On the lowest level, the library gives the possibility to send messages in provided methods, yet on the other hand, it also provides a very high-level `PyGame Engine` introducing integrations with *pygame* framework [20]. Those include functionalities such as the `GameScene` that can present the game state on the screen, the `InputController` for registering and sending player input to the game server, and components for handling the movement of 2d in-game entities such as the `Physics2dComponent`.

On the client side, as visible in listing 2.9, developers are required to create an instance of `UDPClient` class, that provides methods for connection and communication with the server. High-level integrations with *pygame* can be achieved using an object of type `pylon.Engine`. It is also possible to send events via an instance of `ClientMessageDispatcher`.

Listing 2.9 Example client implementation using *MpGameServer* (source: [19])

```
import sys
import pygame
from mpgameserver import UdpClient, EllipticCurvePublicKey

bg_color = (0,0,200)

def onConnect(connected):
    global bg_color

    if connected:
        # on connection success change the background to green
        bg_color = (0,200,0)
    else:
        # on connection timeout change the background to red
        bg_color = (200,0,0)

def main():
    """
    Usage:
        python template_client.py [--dev]
        --dev: optional, use development keys
    """
    pygame.init()

    clock = pygame.time.Clock()
    FPS = 60
    host = 'localhost'
    port = 1474
    screen = pygame.display.set_mode((640, 480))
    client = UdpClient()
    client.connect((host, port), callback=onConnect)

    while True:
        dt = clock.tick(FPS) / 1000

        # TODO: process events, update game world, render frame, send messages

        screen.fill(bg_color)
        client.update()

        for msg in client.getMessages():
            try:
                # TODO: process message
                print(msg)
            except Exception as e:
                logging.exception("error processing message from server")
```

```
            pygame.display.flip()

        pygame.quit()

        if client.connected():
            client.disconnect()
            client.waitForDisconnect()
```

On the server side, as presented in listing 2.10, it is required to define a custom subtype of a base class `EventHandler`, which will handle events such as client connection or disconnection and received message, and an instance of `ServerContext`. Using objects of the aforementioned types, it is possible to create an object that will accomplish server functionalities. There exist two possible types to choose from, the `TwistedServer` and `GuiServer`, the difference being the latter in addition to standard functions, also provides a Graphical User Interface (GUI) presenting network parameters of the server in real-time.

Listing 2.10 Example of server implementation using *MpGameServer* (source: [19])

```
    import os
    import sys
    import logging

    from mpgameserver import EventHandler, ServerContext, TwistedServer, \
        GuiServer, EllipticCurvePrivateKey

    class MyGameEventHandler(EventHandler):

        def starting(self):
            pass

        def shutdown(self):
            pass

        def connect(self, client):
            pass

        def disconnect(self, client):
            pass

        def update(self, delta_t):
            pass

        def handle_message(self, client, seqnum, msg):
            pass

    def main():
        """
        Usage:
            python template_server.py [--gui]
            --gui : optional, display the metrics UI
        """

        # bind to an IPv4 address that can be accessed externally.
        host = "0.0.0.0"
        port = 1474
        key = None
        logging.basicConfig(level=logging.DEBUG, format='%(asctime)-15s %(levelname)s
%(filename)s:%(funcName)s():%(lineno)d:%(message)s')
        ctxt = ServerContext(MyGameEventHandler())

        # command line switch controls running in headless mode (default)
        # or with the built-in gui server
```

```
        if '--gui' in sys.argv:
            server = GuiServer(ctxt, (host, port))
        else:
            server = TwistedServer(ctxt, (host, port))

        server.run()
```

Strengths of *MpGameServer*:

- Has powerful high-level integrations with one of the most popular libraries for game development in Python, *pygame*.
- Has built-in handling of object movement.
- The library has a high-performance goal of handling 128 concurrent clients with 60 server ticks per second.
- Has very extensive reliability and security features, such as packet encryption, Distributed Denial of Service attack mitigations, and optional guaranteed datagram delivery.
- Is still actively developed with a large road map of functionalities to come.
- It is open-source and free for commercial use (LGPL-2.1 license).
- Is publicly available on PyPI.

Weaknesses of *MpGameServer*:

- The library lacks high-level features that could simplify development and boost productivity, such as synchronized shared game state and game lobby.
- Due to its complexity, this library has the highest entry-level out of all analysed solutions.

2.2.5 Panda3D Distributed Networking

As described in official *Panda3D* documentation [21], it is an open-source 3D engine made for the development of video games and other visual art, which can be used with either Python or C++. Considering multiplayer, it provides two APIs developers can use to introduce networking functionalities into their games, a low-level *Datagram Protocol* for sending developer-defined packets via UDP or TCP, as well as a high-level *Distributed Networking*, which works by creating distributed objects that are automatically synchronized across the game server and all connected clients [22].

The core feature of *Panda3D Distributed Networking* are the *Distributed Objects*, which as mentioned in the official documentation [22], are the entities managed in the distributed system. Most objects require two implementations, one for the client, and the other for the server. An example of a client-sided *Distributed Object* implementation can be found in listing 2.11. What can be seen, is that *Panda3D* library introduces a `DistributedObject` base class, that can be easily inherited by the actual implementation. Additionally, methods named with `d_*` prefix are treated as distributed methods that will send messages to the other side of the system.

Listing 2.11 Implementation of a client-sided *Distributed Object* in *Panda3D* (source: [22]).

```
from direct.distributed.DistributedObject import DistributedObject

class DGameObject(DistributedObject):
    def __init__(self, cr):
        DistributedObject.__init__(self, cr)

    def d_sendGameData(self):
        """ A method to send an update message to the server.  The d_ stands
        for distributed """

        # send the message to the server
```

```
        self.sendUpdate('sendGameData', [('ValueA', 123, 1.25)])
```

Strengths of *Panda3D Distributed Networking*:

- It is a high-level solution tailored specifically to the needs of the *Panda3D* game engine.
- Its implementation of distributed objects removes many responsibilities from the game developers, simplifying the process and reducing the required work.

Weaknesses *of Panda3D Distributed Networking*:

- It is framework-specific, as it is usable only with *Panda3D*.
- It is required to provide separate client-sided and server-sided implementations of *Distributed Objects*.

2.3. Conclusions

The analysis of the existing solutions for engines implementing network functionalities of online multiplayer games presents a few areas of possible improvements. As of now, no framework supports a fully automated synchronized share game state. Even though the *PyGaSe* library offers similar functionality, its open design moves the responsibility of handling ownership of certain in-game entities and creating methods proper for accessing objects to the developer. To add to that, currently, the only solution that has in-built lobby functionality is the Simple-Game-Server package, which lacks other features and is prone to cheating due to being entirely client-authoritative. What also should be acknowledged is that none of the engines mentioned previously provide functionalities enabling the analysis of the game state in real time. The *MpGameServer* library offers a graphical user interface presenting data related to network performance, yet there is no possibility of seeing a snapshot of in-game objects and variables, which would be beneficial not only for debugging purposes but also for the analysis of players' behaviour and gameplay patterns.

The author concludes that there is a place in the market for a solution that would put programmers' satisfaction and productivity as the highest priority, lowering the high entry point into online multiplayer game development. Such a solution should offer an easy-to-use complete game state management system, reducing the amount of code necessary to handle both client and server functionalities, simultaneously providing high-level features for handling object movement, lobby, and the possibility to access game state snapshots in real-time for analysis and debugging purposes.

3. Concept of a high-level framework for networking of multiplayer games

This chapter discusses the concept of a high-level framework that would solve the problem of networking of multiplayer games. It presents the motivation behind the and lists functionalities that such a solution should provide. Additionally, the problem of concurrency in Python is explored and its limitations are pointed out, examining asynchrony and multiprocessing as possible alternatives to threading.

3.1. Motivation

As already mentioned previously, the entry point into the development of online multiplayer games is high due to the required expertise in computer networking. The parallel nature of the topic introduces unique challenges, that could cause unexperienced developers to abandon their dream projects, or even not to start them at all. Those pains cannot be eliminated entirely, however, a high-level framework can make a significant difference. Such a solution can take care of the advanced network functionalities offering an easy-to-use interface. This without any doubt can enable novice programmers to do things they thought were impossible at their current level of expertise. Additionally, it would significantly boost productivity and increase work satisfaction for skilled practitioners.

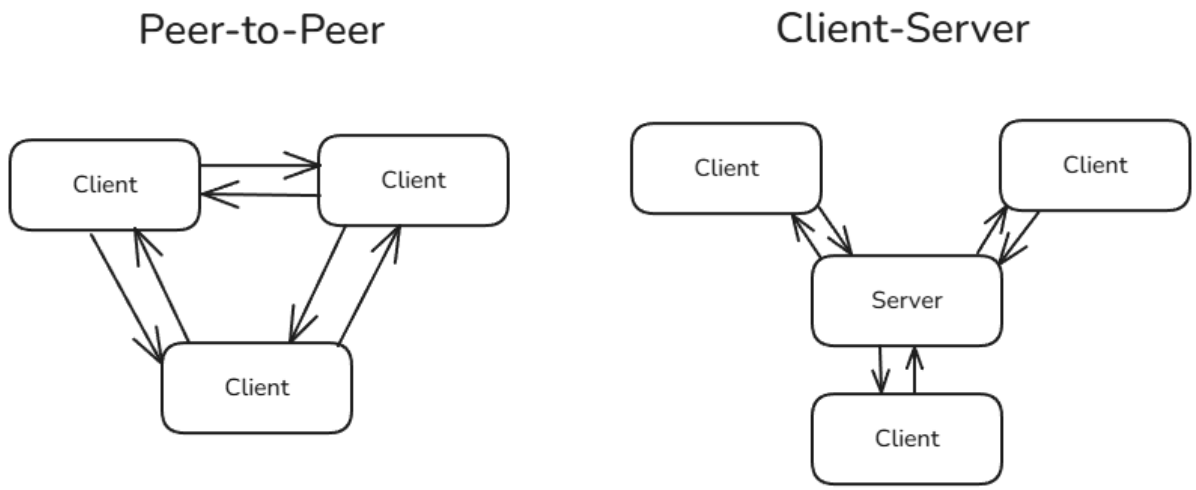

Figure 3.1 Diagram presenting P2P and client-server architectures (source: own elaboration).

One of the most significant decisions that needs to be made is the choice of network architecture. The two most notable ones are Peer-to-Peer (P2P) and client-server. As described in [23], P2P is a decentralized model, where all participants communicate with each other directly. This aspect is also depicted in a corresponding diagram in figure 3.1, where each client directly reaches all others. On the other hand, as presented in [32] and figure 3.1, in client-server solutions there is a distinguished server node, which works as a central point of control and data management. When considering the use case of online multiplayer games, the dedicated server design is the superior one. It provides more consistent performance and allows for much easier state synchronization than the Peer-to-Peer alternative. This is especially crucial, considering the aspect of cheating, which is pointed out in [24]. This article emphasizes that in contrast to single-player games, malicious actions affect not only the

player that conducts them but also can be detrimental to the experience of other users. Knowing that such situations are inevitable, it is important not to unconditionally trust clients, as any message could contain a potential exploit. Having an authoritative server, which is a single source of truth that has complete control over the game is a pivotal prevention measure against such actions.

What has to be remembered when designing such a solution is that the development of distributed and centralized applications differs considerably. All of the nodes in the system need to frequently communicate with each other and a shared state needs to be constantly synchronized. The game networking engine should not be just a simple interface for sending messages, which abstracts the complexity of low-level networking. It should also strive to relieve the programmers from the aforementioned responsibilities as much as possible. Each update of the game state and an action trigger requires information to be transmitted between the program instances. Defining proper messages and manually sending and receiving them could be overwhelming and lead to additional complexity. Framework's purpose should be to alleviate as much of the pains of online multiplayer game development as possible, ideally taking all possible intricacies away. Thus, making the development more similar to the programming of a centralized system. To state an example, an enemy non-player character might attack the player, who as a result takes damage and loses a few of their health points. In a single-player game, the change of state would happen simply in a single process via e.g. a change of a variable value. However, in an online multiplayer this update needs to be propagated to all connected client instances, which significantly increases the complexity of achieving a relatively simple functionality. The idea behind the library is to discover places where not only low-level networking but the entire communication aspect can be abstracted from the programmer to provide an experience similar to that of the development of a single-player game. Mirror Networking library took this path and the noteworthy amount of downloads each year mentioned in section 2.1 can be treated as evidence that such design resonates with developers.

Lastly, the framework should also provide a convenient way to analyse a game state and actions such as new client connections in real-time and historically. There are a variety of use cases where such data could be especially useful, one of the most important being the analysis of gameplay patterns. By discovering e.g. what are the positions most frequently occupied by players or what are the items or army units that are rarely used, the developers can identify deficiencies in their games and what areas are worth improving. The data can also be utilized by server administrators to mitigate cheating. Combined with a detailed knowledge of the game mechanics, it will enable the detection of unfair parameter value changes. The state analysis is not only helpful for already deployed games but can also be handy during development by providing insights that can simplify debugging.

The framework characteristics pointed out in this section paint a picture of a high-level tool strongly centred around improvements in developers' satisfaction and productivity. The abstraction of the majority of communication and synchronization logic aims at lowering the entry level for beginners and allowing experienced programmers to put greater focus on the developed game itself. The success of Mirror Networking suggests that such solutions indeed fulfil market needs. The research of existing Python libraries presented in section 2.2 of the thesis showed that there isn't a complete solution on the market yet, which opens up an opportunity for a new engine. One that, by implementing all the listed ideas will attempt to substantially simplify the development of online multiplayer games.

3.2. Expected functionalities

After the analysis of the existent solutions and considering the motivation stated in section 3.1, the following list of expected functionalities of a Python framework for online multiplayer games was concluded:

- **A fully automated synchronized game state**: The most essential feature of the engine. The goal is to fully relieve the programmer from the need for manual propagation of updates across

all the nodes in the system. The framework should also take care of ownership of the elements of the game state to make it simple for developers to forbid players from acting on objects owned by other players and the server. Moreover, it ought to be possible to easily retrieve references to all shared objects.

- **Shared objects:** The library should provide a possibility to create shared objects, such that an instance of them would be automatically created and maintained on all of the clients and the server. Python is an object-oriented language, hence such a possibility would enable developers to create a single class to implement all functionalities of a given in-game entity, as they would be when developing a single-player game.
- **Automatically synchronized variables:** The developers should have the possibility to create members in the shared classes whose value updates will be transmitted to all connected instances without the need for any programmer's intervention. Such a possibility will greatly reduce the effort to implement various in-game mechanics, such as e.g. gathering resources or taking damage in battles. What has to be taken into consideration, however, is that allowing changes of such variables on the client side could be exploited by cheaters. To prevent tampering from malicious users, value exchanges should be done only on the server and the fields should be read-only on player instances.
- **RPC methods in shared objects:** One of the aims of the framework is to allow programmers to share the client and server code in the same classes as much as it is possible. An important functionality that would play a vital role in fulfilling this objective is the possibility to define methods that will work as remote procedure calls (RPCs). RPC is a communication protocol used to call certain procedures on a remote system, in the same way it would be executed locally [25]. The developers should be allowed to define methods in shared classes that will be triggered on the server side and will be executed on the client side with parameters provided on the server and vice versa. Doing so not only makes it possible to store all functionalities, both those meant to run on a server and on a client, in a single class but also significantly reduces the necessary work, as message serialization and call on receival will be done entirely by the framework.
- **A built-in game lobby functionality:** The lobby is a vital part of online multiplayer games, as in this state it is possible to gather the required amount of players to start a game. The programmers should have access to the current number of connected clients and should be able to easily move from the lobby to the actual gameplay.
- **Client-server architecture:** The engine should support the client-server architecture, with an authoritative server as the single most important source of truth. Following such a design can simplify proper synchronization between all the nodes in the system, provide a more consistent performance, and play a significant role in cheating prevention.
- **Classes for client and server functionalities:** The library should contain classes fulfilling the client and server tasks, that can easily be used in the corresponding implementations. Their purpose would be to provide an easy-to-use interface allowing them to handle connections and disconnections and to access the synchronized game state. The objective is to abstract the low-level networking aspects from the developer, requiring as little domain knowledge as it is feasible.
- **Possibility to choose transport layer protocol:** Considering that response times are essential for fast-paced action games, the User Datagram Protocol (UDP) should be used as a default transport layer protocol, as it is generally faster than the Transport Control Protocol (TCP) [26]. On the other hand, the engine should be universal and should not enforce any particular genre. In the case of strategy games, the speed is not as essential, yet lost packages could have a more detrimental effect. Due to that, a convenient solution would be to leave developers the option to choose between UDP and TCP based on their needs.

- **A possibility to adjust *tick rate* and *update rate*:** Even though the main objective is to create a simple interface for inexperienced developers, it is necessary not to forget about accomplished programmers, who could be searching for a solution that would boost their productivity. Therefore it is necessary to enable the modification of advanced parameters such as *tick rate* and *update rate*. As described in Unity documentation [27], *tick rate* is the frequency of server game state updates, whereas *update rate* is the frequency of clients sending and receiving data to and from the server. Both values are measured in hertz. What is emphasized in [27], is that a higher *tick rate* tends to increase the responsiveness of gameplay for players, but on the other hand, it leads to more computations on a game server. Additionally, a higher *tick rate* wouldn't be beneficial if the *update rate* parameter wouldn't be similarly large. Considering this importance, allowing developers to modify those parameters provides a crucial optimization tool for proficient users.
- **A built-in solution for the movement of in-game entities:** As the movement of objects is a common case for any game, not only online multiplayer ones, another high-level feature of the solution should be a dedicated class for handling position in dimension space. Such type should be available to use as a member in the shared classes, and the engine should automatically detect that a given object contains such a field.
- **A database integration:** The framework should offer the possibility to connect to a database for frequent writing of game state snapshots and events such as a client connection to the server. Such functionality would allow both real-time and historical access, enabling developers the ability to analyse gameplay patterns, identify cheating behaviours, and debug the game.

3.3. Limitations of Python concurrency

The developer experience alone is not enough for a solution to be useful, as sufficient performance is a core necessity. As the goal is to create a solution specifically in Python, its limitations need to be taken into consideration during the design phase. Handling communication with a large amount of clients is a task suited for the introduction of concurrency, therefore a crucial critical caveat of multithreading in this programming language, the *Global Interpreter Lock* (GIL), needs to be addressed.

As described in the 15th chapter of "Mastering Concurrency in Python" by Quan Nquyen [28], GIL is a global lock for the Python interpreter, that must be obtained by any instruction to execute CPU-bound tasks. This ultimately allows only a single thread to run at a time, with a notable exception of I/O-bound tasks. The reason for such restriction is rooted in the memory management of the language, to be exact in the reference counting mechanism that is used to manage the lifetime of objects.

Listing 3.1 The native implementation of Python object (source: [29])

```
typedef struct _object {
        unsigned int ob_refcnt;
        struct _typeobject *ob_type;
} object;
```

This topic was thoroughly discussed by one of Python's contributors, Larry Hastings in his speech "Python's Infamous GIL" during the PyCon conference in 2015 [29]. The native implementation of a Python object, which is presented in listing 3.1, contains at least two members:

- a pointer to a struct containing various characteristics of the type;
- an unsigned integer value representing the number of existing references to the instance;

This reference counter needs to be frequently incremented and decremented in order to properly reflect the current runtime state. What is key to acknowledge, is that even though those operations seem to be atomic from the abstraction of a modern programming language, in fact on the machine code level they are not. This presents a challenge for multithreaded programs, as the references would be added and deleted from many threads running concurrently, which could result in race conditions. Such issues are unacceptable in such essential functionalities of a language as object lifetime management, therefore preventive measures had to be implemented. As mentioned in [28], *Global Interpreter Lock* was chosen as the solution, due to being the most efficient and easiest to create out of all the possible implementations that were considered in the early stages of the development of Python.

The direct result of the existence of GIL is the lack of significant performance gain out of using multithreading. An experiment conducted in [28] suggests that although there is a slight improvement in comparison to sequential programs, it is not significant. Considering that the traditional threads do not seem to be a viable solution to build the framework on. This points to the use of alternatives, of which there are a few. The most obvious one is to use subprocesses instead of threads, as *Global Interpreter Lock* works only in the scope of a single process. This can be achieved thanks to the *multiprocessing* module, one of the standard libraries. As presented in the 6th chapter of [28], it provides functionalities for starting, managing, and communicating between subprocesses. Thanks to Python's C API, it is also possible to create native extensions in C or C++, which would allow to temporarily release the lock to perform specific operations [30]. Additionally, it is possible to use asynchronous programming to improve the efficiency of single-threaded programs and to take advantage of the fact that I/O-bound tasks can be excluded from the mechanism. Lastly, it has to be mentioned that the Python 3.13 release came with an experimental free-threaded mode, which disables the GIL. Unfortunately as stated in the official documentation [31], currently enabling it significantly reduces single-threaded performances and there still could be unfixed issues, this feature can not be considered a stable solution suitable for the project.

3.4. Comparison of asynchrony and multiprocessing in Python

For the purpose of the project, two threading alternatives, multiprocessing and asynchrony, were explored. This section describes performance tests conducted to distinguish the more suitable solution for the use case of a framework for networking online multiplayer games. The following parts present implemented Proofs of Concept (PoC), pointing out their comparative strengths and weaknesses, and presenting results of conducted performance tests.

3.4.1 Multiprocessing with Shared Memory

A game server is required to handle multiple client connections at once, therefore introducing concurrency in the communication with other instances should be beneficial performance-wise. Considering this fact and the limitations posed by GIL, one idea would be to have more than one UDP/TCP server instance. This can be achieved by spawning multiple subprocesses for handling communication, each intended to handle a limited subset of clients. Such architecture is presented in figure 3.1, which depicts a design proposal using multiprocessing, where the mentioned child processes are called *client handlers*. In the case of the implemented PoC, the decision was made to have a separate handler for each connected client.

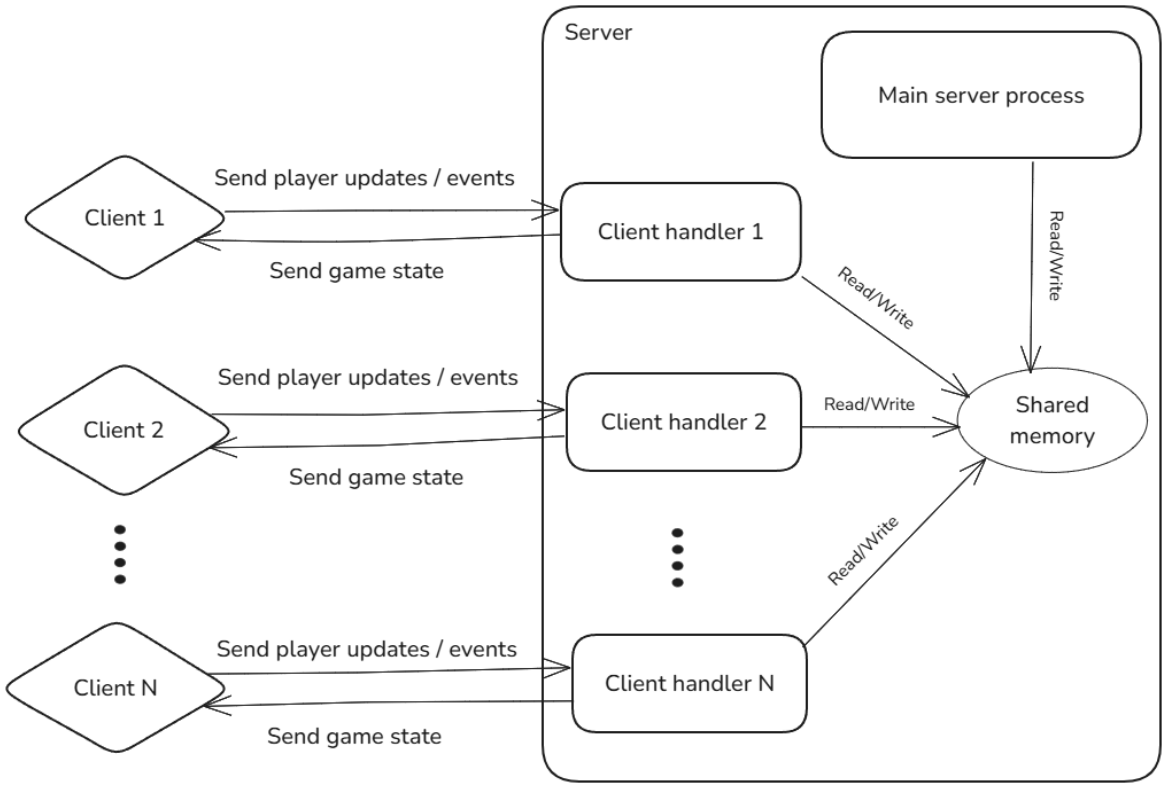

Figure 3.2 Architectural design proposal using multiprocessing with shared memory (source: own elaboration).

The biggest challenge posed by this arrangement is synchronized communication between a main server process, where the game logic would happen, and all of the client handlers. A solution that was used in the PoC is the shared memory provided in the *multiprocessing.shared_memory* module. As stated in the official documentation [32], this library offers the possibility of allocating a shared region of volatile memory, that can be read and written to by started processes. It is also pointed out that this solution has significant performance benefits in comparison to other methods of inter-process communication, therefore the conclusion was that it is most suited to be explored in the experiments.

A crucial functionality of the created Proof of Concept is a `SharedAnyDictionary` class, which is presented in listing 3.2. The *multiprocessing.shared_memory* module offers a few data structures that can be stored in the shared memory, one that was found to be the most convenient was the `SharableList` [33]. Considering that, the idea was to create a map from two such objects, one storing keys and the second storing values. This scheme was materialized in the implementation of `SharedAnyDictionary`, depicted in listing 3.2. For data stored in a game state many times a label, such as health or position is as important as the value itself. Therefore a dictionary was found to be the most suitable for this purpose, enabling easier retrieval of necessary objects. The created class provides an encapsulated interface for such a container and offers methods for the insertion and fetching of data. However, it was not possible to avoid a major drawback. Only selected types can be passed as arguments to spawned subprocess, hence the custom-created class could not have been used. Therefore, even though the lists for keys and values are created as object fields, they need to be passed independently. Due to this reason, all methods for reading and writing are static and operate not on the class members, but on `ShareableLists` taken as parameters.

Listing 3.2 Implementation of `SharedAnyDictionary` from PoC using multiprocessing with shared memory.

```
    class SharedAnyDictionary:
        def __init__(self):
            self.__memory_manager = SharedMemoryManager()
            self.values = None
            self.keys = None

        def start(self):
            self.__memory_manager.start()
            self.values = self.__memory_manager.ShareableList([EMPTY_VALUE for _ in
range(10000)])
            self.keys = self.__memory_manager.ShareableList([EMPTY_VALUE for _ in range(10000)])
            print('Shared AnyDictionary started')

        def __del__(self):
            self.__memory_manager.shutdown()
            print('Shared AnyDictionary has shutdown')

        @staticmethod
        def get_next_free_key(keys: ShareableList) -> Optional[int]:
            for i, v in enumerate(keys):
                if v == EMPTY_VALUE:
                    return i
            return None

        @staticmethod
        def contains(keys: ShareableList, key_to_find: str) -> Optional[int]:
            for i, v in enumerate(keys):
                if v == key_to_find.strip():
                    return i
            return None

        @staticmethod
        def insert(keys: ShareableList, values: ShareableList, lock: Lock, key: str, value: str):
            with (lock):
                try:
                    index = SharedAnyDictionary.get_next_free_key(keys) if not \
                    SharedAnyDictionary.contains(keys, key) else \
                    SharedAnyDictionary.contains(keys, key)
                    keys[index] = key
                    values[index] = value
                except TypeError:
                    print('Shared AnyDictionary full!')
                except ValueError:
                    print('Item size exceed memory size of Shared AnyDictionary!')

        @staticmethod
        def read(keys: ShareableList, values: ShareableList, lock: Lock, pid: int):
            with lock:
                print(f'Process {pid} reads -> {[(k, v) for (k, v) in filter(lambda tup: tup[0]
!= EMPTY_VALUE, zip(keys, values))]}')
```

Strengths of a solution based on multiprocessing with shared memory:

- It is possible to achieve true concurrency and avoid the drawbacks of the *Global Interpreter Lock*.

Weaknesses of a solution based on multiprocessing with shared memory:

- The `SharableList` [33] data structure comes with various drawbacks. There is no possibility of changing its length, therefore the size would need to be allocated in advance. In cases where the initial prediction of the maximal number of items would be incorrect, this would be a major problem. Additionally, only a subset of simple data types, namely *int, float, bool, str, bytes* and

None can be stored in this structure. Due to that a sophisticated logic would need to be additionally implemented to store complex types, such as classes.

- This architecture requires highly efficient hardware on the server side, as multiple Python processes would need to be started on a server to handle a high number of client connections.

3.4.2 Asynchronous Programming

The other proposed solution is based on asynchronous programming, a technique where as described in [34], a unit of work can run apart from the main application thread. This technique is advantageous when dealing with I/O operations, as it allows other tasks to be executed while, for example, waiting for a response after sending a request. It is especially important, considering that as mentioned in section 3.3, tasks of such kind can be excluded from the *Global Interpreter Lock*. This could provide additional processing time for other necessary functionalities.

The proposed architectural design for a framework using asynchrony is depicted in figure 3.2. There exists a single server process with distinct threads for asynchronous communication with clients and game logic. The reasoning behind the usage of traditional threading in this scenario is to create a clear separation of the asynchronous code from the program written by a library user. As it is rather unlikely that the server-side game logic functionalities will require operations that are not CPU-bound, it is assumed that it will be implemented in a synchronous way. Considering that, the objective is to completely abstract the asynchrony from the developer, making sure it does not affect the way they would program their games.

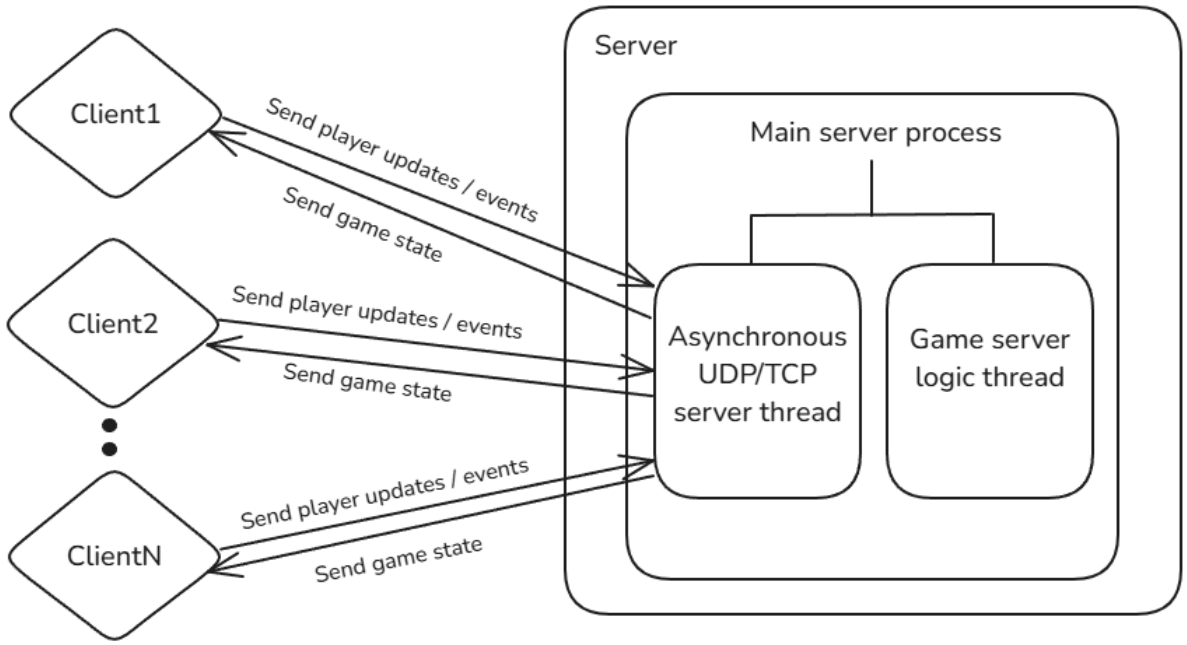

Figure 3.3 Architectural design proposal using asynchronous programming (source: own elaboration).

For the Proof of Concept, an asynchronous UDP server was created using the *asyncio* and *asyncudp* packages [34], [35]. Its implementation is visible in listing 3.3. As mentioned in [34], *asyncio* is a Python module that intends to make asynchronous programming more accessible. It offers constructs, such as the *event loop* that runs and manages asynchronous tasks and *coroutines*, which are special functions that can be run in the loop. Examples of *coroutines* can be found in listing 3.3, namely the methods `run_async_server_udp()`, and `modify_game_state()`, which are defined with

the *async* keyword. The first of them is executed by the `asyncio.run()` function, which is as described in [36] is responsible for calling passed *coroutines* and managing the *event loop*.

Listing 3.3 UDP server implementation from the PoC using asynchronous programming.

```
class AsynchronousServer:
    def __init__(self):
        self.game_state: dict[str, any] = {}
        self.lock = Lock()

    async def modify_game_state(self, message):
        for k, v in message.items():
            async with self.lock:
            self.game_state[k] = v

    async def run_async_server_udp(self):
        socket = await asyncudp.create_socket(local_addr=('127.0.0.1', 8888))

        while True:
            data, address = await socket.recvfrom()
            message: dict = pickle.loads(data)
            await self.modify_game_state(message)
            socket.sendto(pickle.dumps(self.game_state), address)

    def start_server(self):
        asyncio.run(self.run_async_server_udp())
```

Strengths of a solution based on asynchronous programming:

- There are no limitations to the data structures that could store the game state.
- As there would be only a single server process instead of multiple in the solution using multiprocessing, it is expected that the resource usage will be lower.

Weaknesses of a solution based on asynchronous programming:

- As only a single thread can be used, the solution would not be able to fully utilize the computational power of modern multicore processors.
- Considering that games using the framework would be implemented in a synchronous way, additional effort needs to be made in order to fully abstract asynchrony from the developers.

3.4.3 Performance tests

To decide which architecture is more efficient in the use case of an engine for networking of online multiplayer games, performance tests were conducted on both Proofs of Concept. The goal was to verify how a particular design would handle communication with different numbers of concurrent clients. The defined range of connections was set to be all powers of 2 in the range from 2 to 32. In the test scenario, all of them would send 100 messages containing a serialized Python dictionary imitating a game state. The server would update its copy of the structure and respond to each of them with the same object. A key metric that was chosen for the performance assessment was the average response time, namely the calculated mean of time passed from sending the request from the client to receiving a reply from the server. All instances were running on a single machine to mitigate potential network delays, that could affect the measurements. The time interval between each sent message was set to 15.625ms, which resulted in approximately 64 send client messages per second, which is equivalent to having a 64 Hz *update rate*.

Each test case was repeated three times, and the average results from all experiments are presented in table 3.1. The measured mean response times in the PoC using asynchronous programming are from ~8 up to ~19 times lower than the equivalents from the prototype using

multiprocessing. The average response times of 11.22210ms for 16 clients and 34.14261ms for 32 clients of the latter solution strongly indicate that this architecture is not efficient enough for a video game server handling multiple concurrent players. On the other hand, the measured performance of the solution based on asynchrony is auspicious. As stated in [37], the average latency on a Local Area Network (LAN) tends to be around 1-3ms and between 30-50ms on a Wide Area Network (WAN) on a national scale. The measured response times are placed in the middle of the range for LAN and are notably lower than those for WAN, therefore the conclusion is that the additional delay introduced by the asynchronous server would not significantly affect gameplay responsiveness.

Table 3.1 Performance test results of PoCs using asynchronous programming and multiprocessing (less is better).

Number of clients	Average response time [ms]	
	PoC based on asynchrony.	PoC based on multiprocessing.
2	0.20028	1.65014
4	0.11570	1.53939
8	0.25885	2.34008
16	0.80258	11.22210
32	1.74603	34.14261

Considering that a mean average does not provide insight into the distribution of measured values, it is also valuable to analyse measurements on a histogram plot, which is included in figure 3.3. The data on the chart was gathered from a single run of a test case with 32 clients. What is clearly visible, is that the vast majority of response times were close to 0ms, which is a positive indicator of the performance. There were however a few outliers, with some measured values being close to 15ms, which is significantly worse than the calculated average of 1.74603ms, yet considering the noted average WAN latency, it would still be acceptable if isolated messages were sent with such a delay.

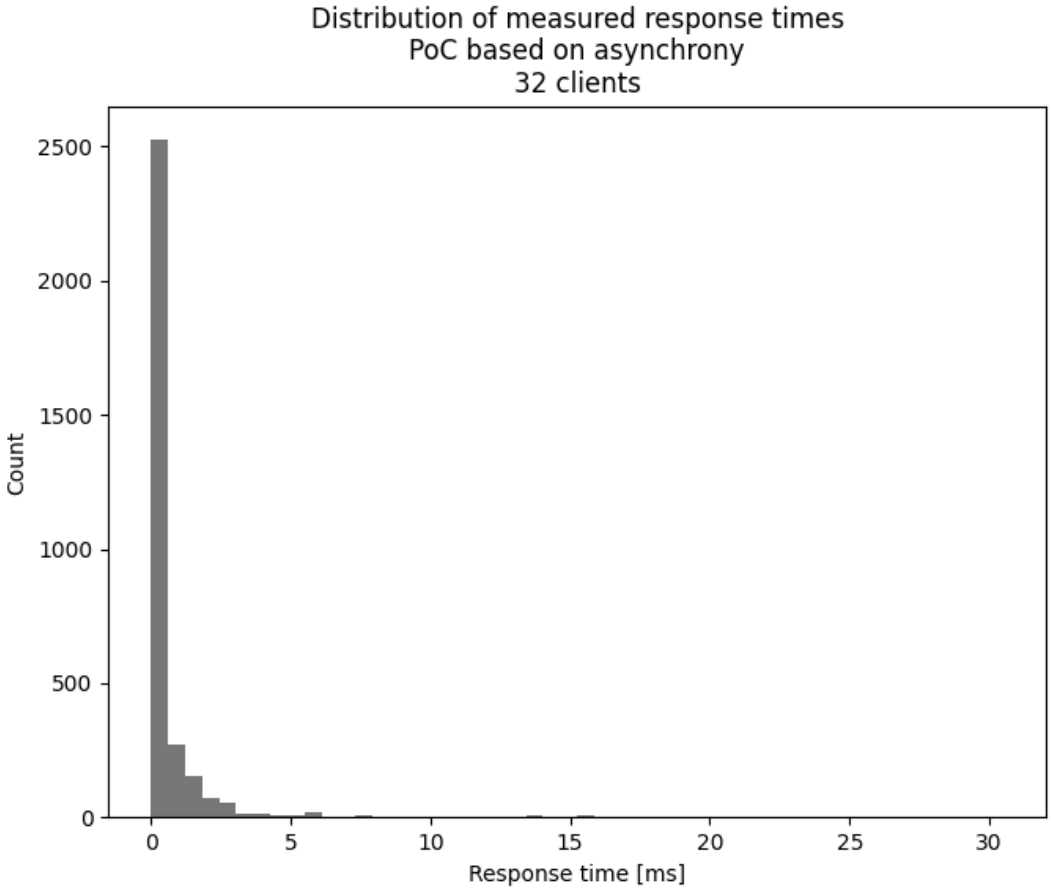

Figure 3.4 Histogram chart presenting the distribution of measured response times for PoC based on asynchrony from a single trial of the test case with 32 clients.

3.4.4 Conclusions

After thoroughly examining the strengths, weaknesses, and performance, a decision was made to follow the proposed architecture based on asynchronous programming. The efficiency of such a solution was tested to be significantly higher than the one based on multiprocessing with shared memory, which is the strongest reason behind this design choice. Moreover, limitations of available data structures imposed by the *multiprocessing* module would entail the implementation of non-trivial logic for handling complex types, such as classes in the game state. A solution based on asynchrony would be free of such restrictions, which would greatly simplify the development of the library.

4. Used technologies

This chapter describes technologies used during the implementation of the engine prototype. It presents a brief overview of the Python programming language and used libraries, as well as a short description of the MongoDB database and other important tools used throughout the prototype development process.

4.1. Python programming language

Python is a high-level interpreted programming language, created by a Dutch software engineer, Guido van Rossum. As can be read in his foreword to a book titled "Programming Python" [38], the idea behind Python was to create a language inspired by ABC, a teaching language aimed at beginner programmers, that would also appeal to more experienced developers using Unix/C. A strong focus was placed on code readability and ease of use. Thanks to that, Python is now generally regarded as an adequate choice for a first programming language, due to being highly learnable. As admitted by Van Rossum himself in [38], one of the weakest aspects of the language is its relatively weak performance. However, in return, the simplicity of Python significantly reduces the required coding and debugging time, which increases programmers' productivity and accelerates the delivery of software. On the other hand, as mentioned in section 3.3, it is possible to write Python extensions in C or C++, which allows for the interpreted language to serve as a convenient interface for developers when all heavy computations are done in native implementations. Those characteristics resonate greatly with modern programming audiences, as Python is placed firmly in first place in the annual language popularity ranking of the IEEE Spectrum magazine [39]. Crucial factors influencing that are well-known libraries for usage in fields such as Artificial Intelligence (AI) and machine learning (ML) and the wide usage of the language for educational purposes.

Figure 4.1 Screenshot from the "Yawnoc" video game (source: [40])

When discussing the topic of video game development, Python is usually not often brought up, as generally compiled languages such as C++ and C# are considered to be more suitable for this task. This does not mean, however, that no video games are being created in this language. It might be more of a niche in comparison to other use cases mentioned previously, but there are still tools such as e.g. *pygame*, described more deeply in section 4.5. It is not as powerful as more mainstream counterparts such as Unreal Engine and Unity Game Engine, yet still allows the creation of moderately advanced games. One example of such can be seen in figure 4.1, which presents a screenshot from a "Yawnoc" video game. It was made using this module by an independent game developer and content creator identified with the nickname "DaFluffyPotato" [40]. This programmer is recognizable in the community due to his YouTube channel [41], where he posts videos about game development using Python. His content is a great example proving that even though this interpreted language may not be as fast as its compiled counterparts and the available tools are not as comprehensive as the most popular engines on the market, it is still possible to create visually pleasing, complex games using it.

4.2. asyncio and asyncudp libraries

As mentioned in section 3.4.2, Python standard libraries contain the *asyncio* module, which intends to make asynchronous programming more accessible. Apart from functionalities allowing enabling code development following asynchrony, the package comes with *streams*, which as described in the official documentation [42] are primitives for handling network connections. These provide a high-level interface for communication abstracting low-level protocols from the programmer. The downside is that *streams* support TCP only, hence they cannot be used with UDP, which was chosen to be the default protocol for the framework. Therefore to handle communication using the latter protocol the *asyncudp* package was chosen [35]. The main factor behind this decision was the simplicity of use of the package. As it can be seen in listing 4.1, which presents an exemplary implementation of a UDP server using this module, the amount of required code lines is low, the interface is clear and simple, and additionally, there is easy access to the sender's IP address which can be used e.g. for player identification.

Listing 4.1 Example of UDP server implementation using *asyncudp* module (source: [35]).

```
import asyncio
import asyncudp

async def main():
    sock = await asyncudp.create_socket(local_addr=('127.0.0.1', 9999))

    while True:
        data, addr = await sock.recvfrom()
        print(data, addr)
        sock.sendto(data, addr)

asyncio.run(main())
```

4.3. pickle and msgpack libraries

To send messages between clients and a server, it is necessary to serialize them into a transferable byte stream format beforehand. As mentioned in the official documentation [43], using *pickle* standard module is a preferred way of achieving it in Python. This is a very convenient solution as it allows not only the serialization of built-in types but also instances of custom classes. However, it also has its drawbacks which were pointed out in the article "Don't Pickle Your Data" [44]. The major problem is the module's performance compared to other solutions, both in terms of speed and size of

serialized objects. The results of the experiments conducted by the article's author presented in figure 4.2 show how significant those differences are, especially in terms of number of serialized items per second. Another downside of the *pickle* module mentioned in [44] is its security risk. Deserialization of malicious data using this library can lead to security breaches, such as arbitrary code execution. Due to this fact, it is strongly emphasized to only *unpickle* data from trusted sources.

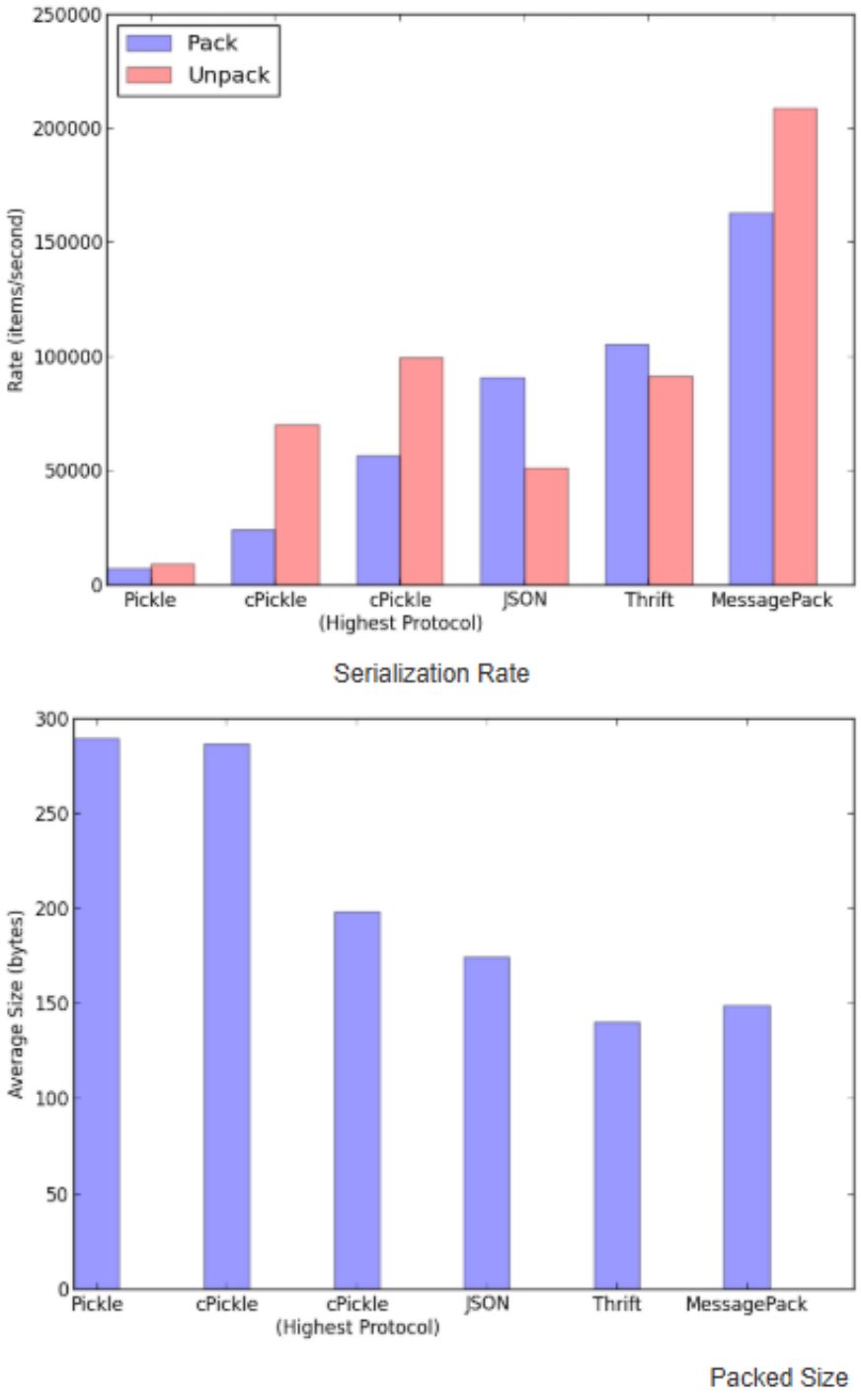

Figure 4.2 Charts comparing the rate of serialized items per second and the average size of serialized items of different solutions available in Python (source: [44]).

Those two mentioned issues of the pickle package are especially problematic in the use case of the framework for networking online multiplayer games. The server needs to handle the serialization and deserialization of messages for multiple concurrent clients, therefore the performance is a crucial factor. On the other hand, as the data received by the server will be produced on remote player instances, there is no easy way to ensure that no tampering was done to it. Potential security exploits on the server side can lead to devastating consequences, including cheating and data breaches. Considering those problems the decision was made to use an alternative to pickle, *msgpack*, as a main solution for serialization and deserialization of messages in the framework. As presented in figure 4.1, it has significantly greater rate efficiency than the counterpart from Python standard modules and also produces smaller serialized messages. Additionally as mentioned in the article [45], it does not allow for arbitrary code execution, increasing security. However, as this module does not allow to pack custom classes, *pickle* was also used in the engine prototype, which is described thoroughly in section 5.2.2.

4.4. MongoDB database

As described in [46], MongoDB is a NoSQL, open-source database, which provides a flexible document-oriented way of storing data. Records are kept in structures named *collections*, which can contain multiple documents following the JavaScript Object Notation (JSON) format. This organization gives a lot of freedom to developers, as there is no need for predefined schemas and JSON is suitable for storing complex, hierarchical structures. Thanks to that, MongoDB is a fitting choice for storing game state in the framework, as objects and their fields can be relatively simply mapped into a JSON document, in a format that is easy to understand. It also relieves programmers of the additional effort required to define schemas, as it is necessary for relational databases.

Another useful feature of this database is MongoDB Compass, which as mentioned in the official documentation [47] is a GUI that allows data querying and analysis. Such functionality can be useful in the framework's use case, as it will provide a convenient way of accessing data, especially for novice developers. Still, it can also relieve experienced programmers from the need to create for example custom scripts for fetching necessary information.

4.5. pygame

As described in [20], *pygame* is a Python module intended for video game development, written on top of the Simple DirectMedia Layer (SDL) library, which as mentioned on its official website [48], is a cross-platform library, which offers a low-level access to input devices and graphics hardware. The short introduction [20], also states that the framework is highly portable and is supposed to run on most operating systems.

There is no dedicated editor for using *pygame*, as it is common for more complex solutions such as Unity Game Engine or Unreal Engine. The development is done solely via Python scripts. Listing 4.2 presents a simple example created using this library taken from the official documentation, which shows basic functionalities. The most vital aspect of this program is the loop, which is made using the *while* instruction. Each of its iterations corresponds to a single video game frame and is responsible for vital functions, such as player input handling and display updates. For example, the method `pygame.key.get_pressed()` method is used to read which keyboard keys were pressed by a player, `pygame.draw.circle()` function draws a circle on a screen and a call of `pygame.display.flip()` updates presented content based on all actions conducted in the loop iteration [49].

Listing 4.2 Simple usage example of the *pygame* module (source: [49]).

```
# Example file showing a circle moving on screen
import pygame

# pygame setup
pygame.init()
screen = pygame.display.set_mode((1280, 720))
clock = pygame.time.Clock()
running = True
dt = 0

player_pos = pygame.Vector2(screen.get_width() / 2, screen.get_height() / 2)

while running:
    # poll for events
    # pygame.QUIT event means the user clicked X to close your window
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            running = False

    # fill the screen with a color to wipe away anything from last frame
    screen.fill("purple")

    pygame.draw.circle(screen, "red", player_pos, 40)

    keys = pygame.key.get_pressed()
    if keys[pygame.K_w]:
        player_pos.y -= 300 * dt
    if keys[pygame.K_s]:
        player_pos.y += 300 * dt
    if keys[pygame.K_a]:
        player_pos.x -= 300 * dt
    if keys[pygame.K_d]:
        player_pos.x += 300 * dt

    # flip() the display to put your work on screen
    pygame.display.flip()

    # limits FPS to 60
    # dt is delta time in seconds since last frame, used for framerate-
    # independent physics.
    dt = clock.tick(60) / 1000

pygame.quit()
```

The key characteristic of *pygame* often brought up in discussions, such as for example in the article [50], is its easy-to-use, simple API. Thanks to that this library allows quick delivery of projects and is a great tool for beginners who would like to diversify their journey of learning programming by developing video games. Considering that, it was decided that an usage example of the developed framework for networking of online multiplayer games will be created with *pygame*. By doing so, such a small project would not only serve as an entry point to working with the newly developed library but would also present how it could be integrated with this well-known module.

4.6. Git and GitHub

As stated in the article on the official Atlassian website, Git is the most popular Version Control System (VCS) available in the market. It is an example of a Distributed Version Control System (DVCS), as unlike systems like Subversion (known as SVN), the complete version history is not stored in a single place, but each working copy of a repository comes with a full change history [51].

The usage of VCS is vital during the development of more complex projects, such as the created framework for networking of online multiplayer games, as it allows to easily track all introduced changes and makes it possible to easily go back to previous versions in case of arisen problems.

To truly benefit from Git, GitHub was used to store the prototype repository. As described in [52], this platform was designed to provide a hosting service Git repositories, offering a web interface and various tools for collaboration. Those functionalities not only made it possible to access the project from multiple devices and to have a project backup, but also can simplify the process of publishing the solution as an open-source project.

4.7. PyCharm IDE

PyCharm is an Integrated Development Environment (IDE) dedicated to Python programming language, created by JetBrains, which is also famous for developing other well-known IDEs, such as IntelliJ for Java and WebStorm for JavaScript [53]. It offers a variety of different functionalities, including an integrated graphical debugger and integration with Version Control Systems, such as Git. Considering the advantages of this IDE mentioned in [53], such as the ease-of-use, convenient ways to view and modify the code, the decision was made to use PyCharm as a main tool to develop the framework.

5. PyMP engine

This chapter describes *PyMP*, created prototype of an engine for networking of online multiplayer games, presenting its architecture and thoroughly characterizing all implemented functionalities, concluding with a discussion about the framework's strengths, areas of possible improvements, and plans for future development.

5.1. Architectural design

Figure 5.1 presents a high-level diagram of the implemented *PyMP* engine prototype, depicting a simplified architecture of the complete system. As it is shown, the shared game state is one of the most important aspects of the framework. Each node in the system has its local working copy. Thanks to that each client can quickly access its snapshot of the state, without the requirement to obtain it from a server. The updates happen periodically, in intervals specified by the *update rate* parameter. By default, the client will communicate with the server 30 times per second. Requests sent in the framework contain a grouped collection of messages, including ones such as:

- events presented in section 5.2.6;
- RPCs described in section 5.2.3;
- `NetworkVariable` updates outlined in section 5.2.4;
- `Position2D` movement speed changes mentioned in section 5.2.5;

For each send request, the server will respond with updates, which are applied to the copy of the shared game state. Based on the research mentioned in section 3.1, it was decided that the engine should follow the client-server architecture with an authoritative server. Therefore, as shown in figure 5.1, any direct modifications to the shared game state can only be done on the server side. Clients are restricted to reading their local copy and triggering changes via sent messages. This ensures that there will be a single point of truth in the system, and it is a crucial cheating prevention measure.

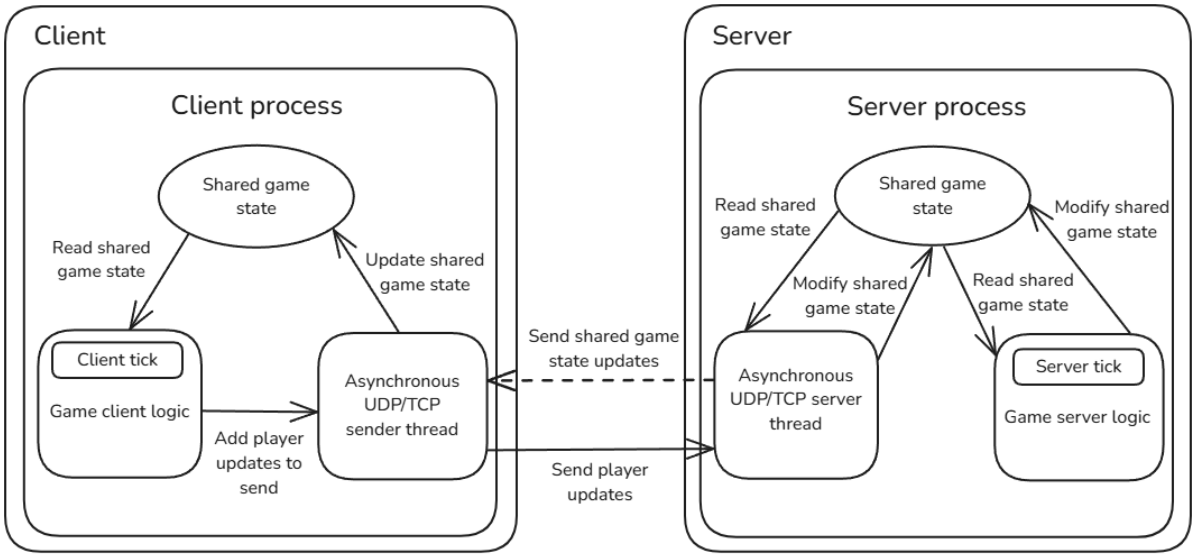

Figure 5.1 A high-level diagram of the *PyMP* engine architecture (source: own elaboration).

The experiments described in section 3.4.3 pointed to using asynchronous programming as an alternative to threading, which would be most suitable for this use case. Therefore the design depicted in figure 5.1 is based on the PoC presented in section 3.4.2, as there is an asynchronous UDP/TCP server running in its dedicated thread on the server and a similar sender on the client side. In this case, the purpose of introducing threading is not to improve performance, but to establish a clear separation between the asynchronous communication code and the most likely synchronous implementation of game logic. Thanks to that, the asynchrony is completely abstracted from the developer.

What is also worth mentioning is that this architecture achieves one of the most important objectives of the framework, as most of the complexity is abstracted from the programmer, and happens automatically without a need for intervention. The only action required to utilize functionalities such as Remote Procedure Calls and the movement of in-game entities, is to call a single function in the game loop, shown in figure 5.1 as a server or client tick.

5.2. Core functionalities

This section presents a detailed description of the core functionalities of the *PyMP* engine, showcasing their implementation and intended use cases.

5.2.1 ServerBase and ClientBase

One of the goals that were outlined in section 3.1 of the work, was to make the development of a multiplayer game close to an implementation of a single-player as much as possible, giving the possibility to encapsulate both client and server functionalities in a single class or a file. However, considering the way video games are likely to be implemented in Python, as showcased e.g. in a simple usage example of *pygame* module contained in section 4.5, this objective needed to be adjusted. It was concluded there is a need for separate client and server implementations, with their own unique logic and game loops. This does not mean that the goal was not met, which is pointed out in section 5.2.2 presenting shared objects called `NetworkObjects`. Those allow sharing both the client and the server functionalities of a given in-game entity in a single class. Even so, for this particular case, it was decided that creating a single implementation for both sides of the system is not optimal in these circumstances. Bearing that in mind, it was decided that *PyMP* should offer developers two classes, `ClientBase` encapsulating all client functionalities, and `ServerBase` to be used in a server implementation.

Even though the aforementioned classes differ in their purpose and functionalities, their interfaces were designed to be as similar as possible. Therefore to activate both the client and the server, it is required to call the `run()` method of a corresponding object. Listing 5.1 presents the implementation of this function from the `ClientBase` class, as well as the constructor of the object. What can be seen is that the first action conducted in the *run()* method is fetching the `client_id` and `update_rate` parameters from the server. This `client_id` value is of great significance as it is a unique identifier of a connected player. It is used to associate received messages with a given client and to mark ownership over shared objects in the game state, which is described more thoroughly in section 5.2.2. There is also a special server value to identify server-owned objects. On the other hand, the obtained `update_rate` is used in the thread with an asynchronous server to define how frequently requests are to be sent to the server.

Listing 5.1 Constructor and `run()` method of the `ClientBase` class.

```
class ClientBase:
    def __init__(self, communication_protocol: CommunicationProtocol = CommunicationProtocol.UDP,
verbose_logging: bool = False):
        self.__base_type = BaseType.Client
        self.__client_id: Optional[int] = None
```

```
        self.__server_address: Optional[str] = None
        self.__server_port: Optional[int] = None
        self.__communication_protocol: CommunicationProtocol = communication_protocol
        self.__network_object_dict: NetworkObjectDictionary = NetworkObjectDictionary()
        self.__network_object_position_dict: Position2dDictionary = Position2dDictionary()
        self.__messages_to_send: GenericQueue = GenericQueue()
        self.__rpcs_to_execute: GenericQueue = GenericQueue()
        self.__received_events: GenericQueue = GenericQueue()
        self.__server_running: Optional[bool] = None
        self.__game_started: Optional[bool] = False
        self.__async_sender_thread = Thread(target=self.__async_sender_callback)
        self.__lobby_snapshot: dict[int, ClientData] = {}
        self.__player_info: dict[str, any] = {}
        self.__is_ready: bool = False
        self.__verbose_logging = verbose_logging
        self.__update_rate: Optional[int] = None

    def run(self, server_address: str, server_port: int):
        try:
            self.__server_address = server_address
            self.__server_port = server_port
            self.__get_client_id_from_server()
            self.__server_running = True
            self.__async_sender_thread.start()
        except Exception as err:
            raise ServerConnectionError(f'Unable to connect to server, reason: {err}')
```

Listing 5.1 also shows all members of the `ClientBase` class, including:

- a `NetworkObjectDictionary` instance for storing shared objects, described in detail in section 5.2.2;
- a `Position2dDictionary` for storing positions of objects in two-dimensional space discussed in section 5.2.5;
- a queue of Remote Procedure Calls outlined in section 5.2.3;
- a collection of events explained in section 5.2.6;
- dictionaries containing player data that can be used in a game lobby and during gameplay summarized in section 5.2.7;

What requires further elucidation in this part of the work is the `messages_to_send` queue. All of the before-mentioned functionalities require an exchange of information between the client and the server. Therefore there was a need for a data structure that would store all of the messages before sending them to the recipient. This purpose was one of the reasons for the creation of the `GenericQueue` class. Its implementation is depicted in listing 2.5. It contains a single Python list of an arbitrary type and a lock ensuring synchronization of conducted operations. The `GenericQueue` allows adding a single object to the collection and offers the possibility to retrieve and clear the entire queue, either returning a standard list or a serialized object containing all of the elements. The second possibility is used in both `ClientBase` and `ServerBase` to send all messages that were created between requests.

Listing 5.2 `GenericQueue` class implementation.

```
class GenericQueue:
    def __init__(self):
        self.__queue: list[any] = []
        self.__lock: Lock = Lock()

    def dump_all_packed(self, client_id=None) -> bytes:
        with self.__lock:
            dump = self.__queue
```

```
            self.__queue = []
        return pack_client_message({client_id: dump}) if client_id else pack_server_message(dump)

    def dump_all_list(self) -> list[any]:
        with self.__lock:
            dump = self.__queue
            self.__queue = []
        return dump

    def add(self, message: any):
        with self.__lock:
            self.__queue.append(message)

    def empty(self) -> bool:
        return len(self.__queue) == 0
```

All messages in *PyMP* are built as Python dictionaries, each containing a required `type` field with an enum value describing their meaning. This is used in functions dedicated to handling received requests in both `ClientBase` and `ServerBase` classes. An example of such a method from the server-related object can be found in listing 5.3. This method is executed when iterating through the received collection of the client messages, in the asynchronous server thread. Based on the `type` value, different actions are performed with the received data.

Listing 5.3 Handling of a single message received from a client in `ServerBase` class.

```
   def __handle_request(self,  request: dict[str,  any], client_id:  int, address:
Optional[tuple[str, int]] = None):
        message_type: MessageType = request['type']
        match message_type:
            case MessageType.RpcReq:
                self.__rpcs_to_execute.add(request)
            case MessageType.EventReq:
                self.__received_events.add(request['event'])
            case MessageType.ClientIdReq:
                if self.__game_started or self.number_of_connected_clients + 1 >
self.max_client_count:

self.__client_id_request_buffer.add(make_client_id_reject_response(ServerBase.GAME_STARTED_REJECT
ION_MESSAGE if self.game_started else ServerBase.SERVER_FULL_REJECTION_MESSAGE))
                    return

self.__client_id_request_buffer.add(make_client_id_response(self.__next_free_client_id,
self.__client_update_rate))
                self.__client_message_queues_dict[self.__next_free_client_id] = GenericQueue()
                self.__client_status_dict[self.__next_free_client_id] = ClientStatusHandler()
                if self.__use_analytical_database:
self.__database_handler.write_client_connected_to_server_event(self.__next_free_client_id,
address)
                print(f'[ServerBase] Player connected from address: {address} -> assigning id:
{self.__next_free_client_id}')
                self.__next_free_client_id += 1
            case MessageType.ClientLobbyHeartbeatReq:
                self.__client_data_dict[client_id] = ClientData(request['isReady'],
request['playerInfo'], address)
            case MessageType.Position2DUpdateReq:
                self.__network_object_position_dict.update_speed(request['objectId'],
request['newSpeed'])
            case MessageType.ClientDisconnectReq:
                self.__client_status_dict[client_id].set_status(ClientStatus.DISCONNECTED)
                print(f'[ServerBase] Client: {client_id} disconnected from server')
```

Even though the interface of the `ServerBase` is similar to the one of the `ClientBase`, there are significant differences in their implementations. Most notably, in contrast to the latter class, `ServerBase` contains a dictionary of queues with messages to send, where each item corresponds to a single connected client. It also offers the possibility to define parameters, such as the previously mentioned `update_rate` and a limit for the number of connected clients, as well as gives the ability to enable the database integration functionality, which is described in section 5.2.9.

A key aspect of both the `ClientBase` and `ServerBase` are their corresponding `tick()` methods. Those functions take care of the execution of scheduled RPCs, object movement, and checking of the current state of client connections on the server side, which is described in section 5.2.8. This method has to be executed once per each iteration of the game loop. Additionally, developers have access to `ServerBase.tick_rate(tick_rate)` and `ClientBase.fps_limiter(fps_limit)` methods, which allow them to control the number of game loop iterations happening each second. The decision not to include this functionality inside the `tick()` methods was validated by the fact that the *pygame* module offers `pygame.time.Clock` object [54] which serves the same purpose. Therefore encapsulation of this feature in the framework could lead to potential confusion.

5.2.2 NetworkObjects and synchronized game state management

The most vital functionality provided by the *PyMP* engine is the `NetworkObject`. This feature allows the creation of shared objects, which are instantiated and maintained on the game server and all client instances. Knowing that Python is an object-oriented language, it is assumed that in-game entities will likely be created as arbitrary classes. Therefore, it was decided that the most appropriate solution is to create a base class that could be inherited from. Listing 5.4 presents the constructor of this class, showing its members. Each `NetworkObject` has the following important attributes:

- `client_id` of the entity's owner;
- `object_id`, which is a combination of the owner's `client_id`, a name of the arbitrary class and an integer number;
- *base*, which stores a reference to the `ClientBase` object on a client instance or to `ServerBase` on a game server. Having this variable was necessary to achieve functionalities such as RPCs and `NetworkVariables`, described in sections 5.2.3 and 5.2.4;

Listing 5.4 `NetworkObject` base class constructor.

```
class NetworkObject:
    def __init__(self):
        self.__object_id: Optional[str] = None
        self.__client_id: Optional[int] = None
        self.__server_owned: Optional[bool] = None
        self.__is_on_server: Optional[bool] = None
        self.__base: Optional[object] = None
        self.__base_type: Optional[BaseType] = None
        self.__is_alive: bool = True
        self.__is_local_client_owner:  Optional[bool]  = None
```

`NetworkObjects` are stored in a data structure named `NetworkObjectDictionary`, a shared game state focal point. Each connected client and the server have a local working copy of this dictionary in `ClientBase` and `ServerBase` instances, which is used to access stored objects. As shown in listing 5.5, it was implemented as a class with two members:

- a Python dictionary with `object_id` strings as keys, and `NetworkObjects` as values;
- a lock for synchronization of conducted operations;

Apart from the functionality of adding a new object, `NetworkObjectDictionary` gives the possibility to query lists of objects using:

- `object_id`;
- `client_id`;
- a name of an arbitrary class;
- the combination of `client_id` and class name;

This is used in both `ClientBase` and `ServerBase` to provide methods for fetching `NetworkObjects`, giving developers convenient ways of accessing all relevant objects. Another worth-mentioning feature of the `NetworkObjectDictionary` is the `get_snapshot()` method, which returns a copy of the dictionary. It is used for periodical writing of game state entries in a database, which is described in section 5.2.9.

Listing 5.5 `NetworkObjectDictionary` implementation.

```
class NetworkObjectDictionary:
    def __init__(self):
        self.__dict: dict[str, NetworkObject] = {}
        self.__lock: Lock = Lock()

    def add_object(self, key: str, value: NetworkObject):
        with self.__lock:
            self.__dict[key] = value

    def get_object_by_object_id(self, object_id: str) -> Optional[NetworkObject]:
        return self.__dict.get(object_id)

    def get_objects_by_class_name(self, class_name: str) -> list[NetworkObject]:
        return self.__get_object_by_id_part(class_name)

    def get_objects_by_client_id(self, client_id: int) -> list[NetworkObject]:
        return self.__get_object_by_id_part(str(client_id))

    def get_objects_by_client_id_and_class_name(self, client_id: int, class_name: str) ->
list[NetworkObject]:
        return self.__get_object_by_id_part(f'{client_id}_{class_name}')

    def get_snapshot(self) -> dict:
        with self.__lock:
            return copy(self.__dict)
    def __get_object_by_id_part(self, id_part) -> list[NetworkObject]:
        found_objects: list[NetworkObject] = []
        with self.__lock:
            for key, value in self.__dict.items():
                if id_part in key:
                    found_objects.append(value)
        return found_objects
```

As shown in figure 5.1, due to the server-authoritative nature of the framework, it was decided that modifications of the shared game state could only be done on the game server. This includes the creation of `NetworkObjects`. Such a design decision was motivated by the fact that if such objects were to be instantiated on a client side, they would need to be validated to prevent potential cheating. Additionally, considering that developers' satisfaction and productivity were treated with the highest priority, the goal was not to force users to create any schemas for shared classes. Therefore a decision was made to use the *pickle* library for the serialization of arbitrary Python objects. This enables easy serialization of `NetworkObjects` without introducing strict limitations to allowed class members. However, knowing the security risks explained in section 4.3, deserialization of data received on the server could lead to potential risks. Due to those reasons, `NetworkObjects` need to be created on the server side. If an instantiation of such an object was to be a result of a player's activity, the suggested solution would be to use events described in section 5.2.6 to trigger creation on the game server. Such

flow is depicted in figure 5.2, which contains a sequence diagram presenting `NetworkObject` creation activated by a player action. The depicted process can be initiated, for example by a decision to recruit a new army unit in a strategy game. Such a choice has to be acknowledged in the client game instance, and based on it a developer-defined event needs to be passed to the `ClientBase` object, which will send it to the server. There, the local game instance needs to fetch this message from the `ServerBase`, and instantiate the relevant `NetworkObject`. After that, it needs to be registered using the `ServerBase.create_player_owned_network_object()` method. This will execute an internal method, presented in listing 5.6. There a unique `object_id` and owner's `client_id` are assigned to the object and its serialized copy is passed in a defined message format to all client message queues. In the next steps, the local `NetworkObject` instance receives all required dependencies, including a reference to `ServerBase` object, and is at the end inserted into the `NetworkObjectDictionary`. Those instructions will be repeated on each client instance, as the object will receive the `ClientBase` dependency and will be added to the local dictionary copy. When it is done, the client game instance can fetch the `NetworkObject` using the provided methods and can be used by the player in their gameplay.

Listing 5.6 Internal implementation of `NetworkObject` registration in `ServerBase` class.

```
   def __create_network_object_internal(self, network_object: NetworkObject, client_id:
Optional[int]):
        object_id_without_counter = f'{client_id if client_id else
ServerBase.SERVER_OWNED_NETWORK_OBJECT_ID_WITHOUT_COUNTER}_{network_object.__class__.__name__}'
        if self.__network_object_id_counter_dict.get(object_id_without_counter, None) is None:
            self.__network_object_id_counter_dict[object_id_without_counter] = 0

network_object._set_ids(f'{object_id_without_counter}_{self.__network_object_id_counter_dict[obje
ct_id_without_counter]}', client_id)
        self.__network_object_id_counter_dict[object_id_without_counter] += 1
        network_object_to_send = copy(network_object)

self.__send_message_to_all_clients(make_network_object_creation_request(network_object_to_send))
        network_object._set_base(self, self.__base_type)
        network_object._set_self_reference_to_network_variables()
        self.__network_object_dict.add_object(network_object.object_id, network_object)
        position_opt = network_object._get_position_reference()
        if position_opt is not None:
            position_opt._set_dependencies(network_object, self.__base_type)
            self.__network_object_position_dict.add_new_entry(network_object.object_id,
position_opt)
        print(f'[ServerBase] Added new NetworkObject -> NetworkObject dict: {network_object}')
        self.__network_object_dict.add_object(network_object.object_id, network_object)
```

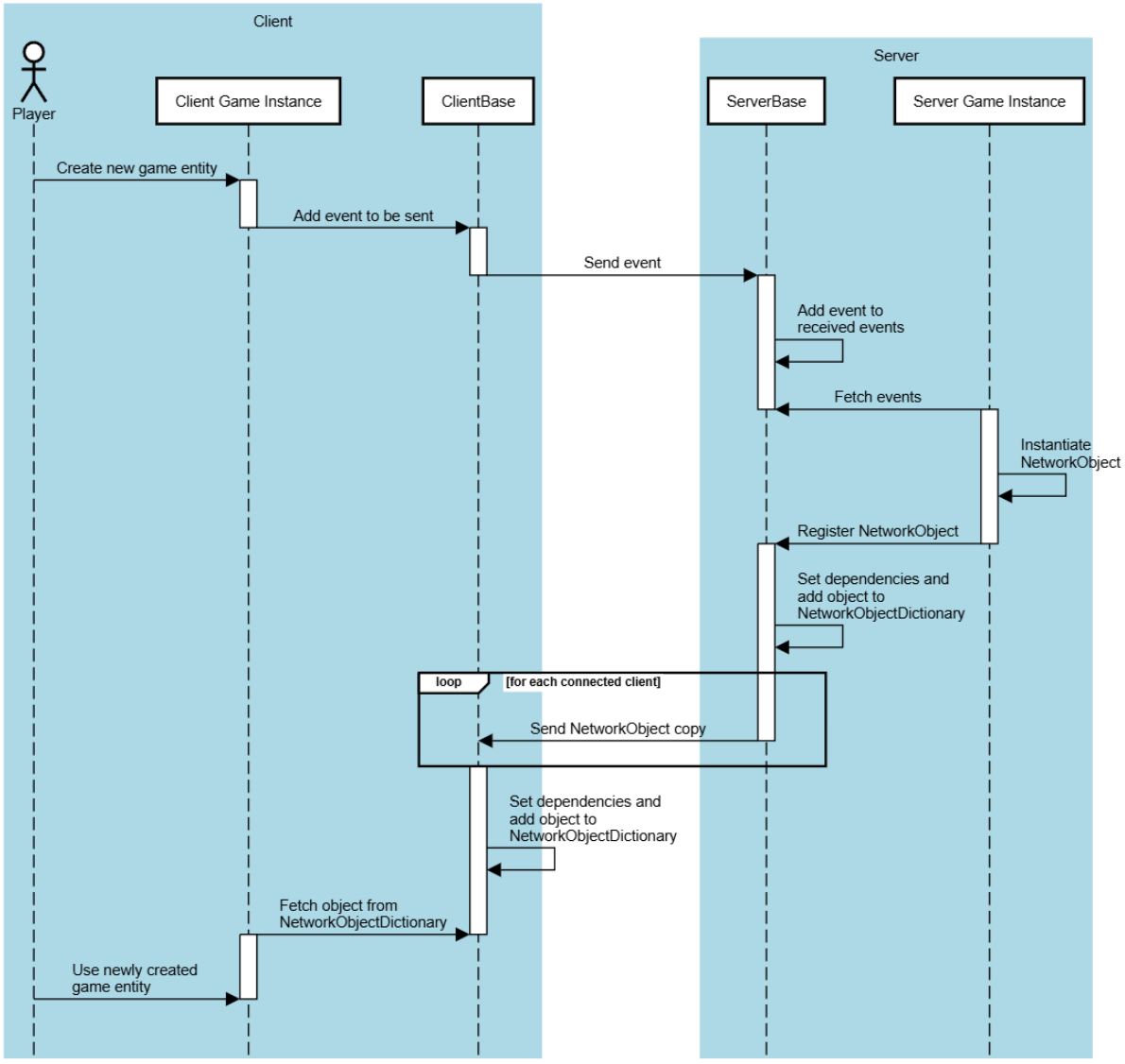

Figure 5.2 Sequence diagram of `NetworkObject` creation triggered by player action (source: own elaboration).

The sequence diagram in figure 5.2 shows a scenario where the process is started on the client side. In case the initiation happens on the server side, the process starts with the step of `NetworkObject` instantiation in the server game instance. An example of such a scenario can be found in lisiting 5.7, which presents the initialization of all in-game entities in a simple example of *PyMP* usage with *pygame*. This method is called in the first game loop iteration and contains the creation of `Player` objects, which are characters steered by a user, as well as `Ball` used in the game and the `Scoreboard`. The actual gameplay in this instance will start when an assigned player object arrives at the local `NetworkObjectDictionary`.

Another important aspect that was tackled during the implementation of *PyMP* is the fact that some objects might have a limited life duration. For example, an army unit might get defeated during a battle. To handle such scenarios, each `NetworkObject` has an `is_alive` property, which is a Boolean value describing if the object should still actively take part in the game. It is not a perfect solution, as the most ideal way of solving this problem would be to delete the instance from the `NetworkObjectDictionary`. However considering that the game instances are to work with fetched

object references from the dictionary, possibly assigned to runtime variables, this was not considered as a viable mechanism.

Listing 5.7 Method for initialization of in-game entities in *pygame* usage example of *PyMP*.

```
   def setup_game(self):
        self.server.create_player_owned_network_object(Player((PLAYER_SIZE_X, PLAYER_SIZE_Y),
100, 180, (10, 790 - PLAYER_SIZE_X), (10, 360 - PLAYER_SIZE_Y)), 1)
        self.server.create_player_owned_network_object(Player((PLAYER_SIZE_X, PLAYER_SIZE_Y),
700, 180, (10, 790 - PLAYER_SIZE_X), (10, 360 - PLAYER_SIZE_Y)), 2)
        self.server.create_server_owned_network_object(Ball(BALL_SIZE, BALL_INIT_X, BALL_INIT_Y,
(10, 790 - BALL_SIZE), (10, 360 - BALL_SIZE)))
        self.server.create_server_owned_network_object(Scoreboard())
```

5.2.3 RPC methods

One of the goals stated in section 3.2 of the work was to provide programmers the possibility to share both the client and the server functionalities in a single class. Game entities are likely to have specific functionalities that are meant to be executed on different sides of the system and often there is a need to initiate the execution of such procedures from a different node. For example, a player's activity might trigger some action that should be done on the game server. To solve this problem, *PyMP* engine introduces the `rpc(<destination>)` function decorator, which can be utilized to use chosen functions as Remote Procedure Calls. Its implementation is shown in listing 5.8. It takes `destination` as a parameter, which is an enum value equivalent to either:

- a single client, owner of the object;
- all connected players;
- the server;

From a client instance it is only possible to trigger RPCs on the server, and equally server can schedule such procedures only on clients. As presented in listing 5.7, there is a conditional check to identify if the current instance is the proper destination of the call. If it is satisfied, the method is called. If not, a defined message containing the method name and parameters is created and added to a relevant queue.

Listing 5.8 `rpc` method decorator implementation.

```
def rpc(destination: MessageDestination):
    def outer_wrapper(func: Callable):
        def inner_wrapper(*args):
            method_name: str = func.__name__
            call_params: list[any] = list(args[1:])
            if args[0]._get_base_type() == destination or (destination ==
MessageDestination.AllClients and args[0]._get_base_type() == BaseType.Client):
                try:
                    func(*args)
                except Exception as err:
                    print(f'[NetworkObject {args[0].object_id}] Unable to execute RPC: {err}')
            else:
                if args[0]._get_base_type() == MessageDestination.Client and not
args[0].is_local_client_owner:
                    raise ForbiddenMethodCall("RPC executed on not owned object")
                try:
                    args[0]._add_rpc_message_to_queue(method_name, call_params, destination)
                except AttributeError as err:
                    print(f'[NetworkObject {args[0].object_id}] Error when adding RPC to message
queue: {err}')
        return inner_wrapper
    return outer_wrapper
```

Listing 5.9 contains a method that is used in `ServerBase` class to execute RPCs, which is called for each such procedure received in a given time interval in the `ServerBase.tick()` method. Firstly a relevant `NetworkObject` is queried from the `NetworkObjectDictionary` by its `object_id`. Then a built-in `getattr()` method is used to obtain the RPC method. Reflective programming is one of the paradigms supported by Python and played a crucial role in this implementation. As described in [55] it is a mechanism that allows the dynamic discovery of the structure of arbitrary objects. The `getattr()` function returns the value of an attribute of a given attribute based on a provided name as a string [56]. Thanks to the fact that methods in Python are also treated as class attributes, this functionality could have been used in this case. To finalize the process, the acquired procedure is executed with a list of parameters attained from a received message.

Listing 5.9 Method for executing RPCs in `ServerBase`.

```
   def __execute_rpc(self, rpc_request: dict[str, any]):
        try:
            network_object: object =
self.__network_object_dict.get_object_by_object_id(rpc_request["objectId"])
            method = getattr(network_object, rpc_request["methodName"])
            call_params: list[any] = rpc_request["callParameters"]
            method(*call_params)
        except Exception as err:
            print(f'[ServerBase] Error when trying to execute RPC: {err}')
```

Usage example of the `rpc` method decorator can be found in a *Knight* class from a simple command-line *PyMP* project, which is shown in listing 5.10. The `attack_random_barbarian()` method is called in a client game instance, after which a proper message is sent to the server and the actual function execution occurs on the game server, with the `attack_type` parameter provided on the player side.

Listing 5.10 `Knight` class constructor and its RPC method from a command-line example of *PyMP*.

```
class Knight(NetworkObject):
    def __init__(self):
        super().__init__()
        self.position = Position2D(0, 0, 0, 0)
        self.weapon = 'sword'
        self.health: NetworkVariable = NetworkVariable(100)
        self.gold: NetworkVariable = NetworkVariable(0)

    @rpc(MessageDestination.Server)
    def attack_random_barbarian(self, attack_type: str):
        if self.is_alive:
            # executed on server
            print(f'Knight {self.object_id} attacking barbarian with {self.weapon} -> attack
type: {attackType}')
            if randint(1, 10) % 2 == 0:
                self.gold.value += 10
                fight_result = 'won'
            else:
                self.health.value -= 5
                fight_result = 'lost'
            self.send_event({"random_battle_result": f'Battle finished -> {self.object_id}
{fight_result} against barbarian!'}, MessageDestination.Client)
```

5.2.4 NetworkVariables

When designing *PyMP*, it was acknowledged that updating shared object variables in a distributed system requires non-trivial work, as the value change needs to be propagated in all nodes via proper messages. `NetworkVariable` was introduced to relieve programmers from this additional work. It is an added type that can be used as a member of `NetworkObjects`, that is automatically synchronized between the game server and all connected clients. Its implementation can be found in listing 5.11. It is meant to serve as a container for a generic type. The value can be obtained and modified using a defined *value* property. Each time the value setter is called, apart from the update of the local value, a defined message is added to communication queues to propagate the change to all connected clients. This was achieved by overriding the Python object's `__setattr__` method, which as mentioned in the related Python Enhancement Proposal 726 (PEP) [57], upon other use cases was meant to allow programmers to intercept value attribute changes in order to update chosen other state. What requires mentioning, is that `NetworkVariables` can only be modified on a server, whereas on the side of a client, they are constrained to be read-only. This was done as a cheating prevention measure, restricting the authority only to the game server. Therefore it was decided that in case of a change attempt in a client instance, an exception will be raised.

Listing 5.11 `NetworkVariable` class implementation.

```
class NetworkVariable:
    def __init__(self, initial_value: T):
        self.__owning_network_object: Optional[T] = None
        self.__attribute_name: Optional[str] = None
        self.__value: T = initial_value

    @property
    def value(self) -> T:
        return self.__value

    @value.setter
    def value(self, new_value):
        if self.__owning_network_object is not None and not
self.__owning_network_object.is_on_server:
            raise ForbiddenMethodCall('NetworkVariable values can only be modified on server')
        self.__value = new_value

    def _set_owning_network_object(self, owning_network_object, attribute_name):
        self.__owning_network_object = owning_network_object
        self.__attribute_name = attribute_name

    def _set_received_value(self, new_value: T):
        self.__value = new_value

    def __setattr__(self, name: str, value: T):
        super().__setattr__(name, value)
        if 'value' in name:
            if self.__owning_network_object is not None and
self.__owning_network_object.is_on_server:

self.__owning_network_object._add_network_variable_update_message_to_queue(self.__attribute_name,
value)
```

As can be noticed in listing 5.11, each `NetworkVariable` stores a reference to the `NetworkObject` that contains it. This was necessary to enable the addition of update messages to queues. To abstract the injection of this value from developers, it happens during the step of setting dependencies to shared objects, which was presented in listing 5.6, which is included in section 5.2.2. Listing 5.12 shows the method of the `NetworkObject` class accomplished this task. Similarly to the process of execution of RPCs described in section 5.2.3, the reflection mechanism is used to discover

all members of the object that are `NetworkVariables` and all found instances receive a `NetworkObject's` self-reference in a dedicated setter method, as shown in the listing 5.12.

Listing 5.12 Method for adding a self-reference to `NetworkVariables` in the `NetworkObject` class.

```
    def _set_self_reference_to_network_variables(self):
        try:
            for attribute_name in filter(lambda name: type(getattr(self, name)) ==
NetworkVariable, filter(lambda name: name[len(name) - 2: len(name)] != '__', dir(self))):
                network_variable: NetworkVariable = getattr(self, attribute_name)
                network_variable._set_owning_network_object(self, attribute_name)
        except Exception as err:
            print(f'[NetworkObject {self.__object_id}] -> unexpected exception: {err}')
```

An example of the usage of `NetworkVariables` can be found in listing 5.10 in section 5.2.3. The *Knight* class has two such members defined in its constructor, `health` and `gold`. Both of those values are modified in the `attack_random_barbarian()` server-side RPC.

5.2.5 Position2D

As the movement of in-game entities is an essential part of many action-based video games, it was decided that the *PyMP* engine should offer a built-in solution dedicated to this functionality. The feature intended to achieve it is the `Position2D`, a type that describes an object's position in a two-dimensional (2D) space, that can be added as an attribute to `NetworkObjects`. The choice of 2D space over a three-dimensional (3D) one was motivated by the fact the *pygame* module is more suitable for the creation of 2D games. Therefore it was assumed that the majority of the projects created using the *PyMP* engine would also be two-dimensional.

Considering the objective of having a single source of truth on the server, stated in section 3.1, no exception was made for the feature of movement as it is also server-authoritative. What it means is that the actual change of position occurs entirely on the server based on speed parameters passed from a client. The entire process of such updates is presented in a sequence diagram in figure 5.3. Based on the player's input, the client game instance should set new velocity values to a `Position2D` member of the local instance of a given `NetworkObject`. Those values will be sent to the server, where the `ServerBase` instance will insert them into the relevant object. The movement itself occurs in the `ServerBase.tick()` method, where the `_move_tick()` method is executed for all existing positions. As seen in its implementation shown in listing 5.13, in this function the received speed values are added to the object's x and y axes coordinates. When it is done, the newly set position is sent to the client, where similarly to the server side, it is set in the local `Position2D` instance in the `ClientBase.tick()` method. When the entire process is completed, the client game instance can access the updated `NetworkObjects` position and render it on a screen in a new location.

Listing 5.13 `Position2D._move_tick()` method implementation.

```
  def _move_tick(self,  collision_boxes=None)  -> (str, (float, float)):
      if self.__base_type == BaseType.Server:
          if self.__received_speed is not None or self.__server_controlled:
              if not self.__server_controlled:
                  if not self.__validate_speed():
                      return None, None
                  self.__speed_x, self.__speed_y = self.__received_speed
                  self.__received_speed = None
              collision_detected = False
              if self.collisions_enabled and collision_boxes:
                  self_collision_box = self.collision_box
                  for box in collision_boxes:
```

```
                    if self_collision_box != box and
self.__check_collision(self_collision_box, box, self.__speed_x, self.__speed_y):
                        collision_detected = True
            if collision_detected:
                return self.__owning_network_object.object_id, self.position
            if self.__x_boundaries[0] <= self.__x + self.__speed_x <= self.__x_boundaries[1]:
                self.__x += self.__speed_x
            if self.__y_boundaries[0] <= self.__y + self.__speed_y <= self.__y_boundaries[1]:
                self.__y += self.__speed_y
        return self.__owning_network_object.object_id, self.position
    elif self.__base_type == BaseType.Client:
        if self.__received_position is not None:
            self.__x, self.__y = self.__received_position
            self.__received_position = None
            if self.__owning_network_object.is_local_client_owner:
                return self.__owning_network_object.object_id, self.speed
            else:
                return None, None
    return None, None
```

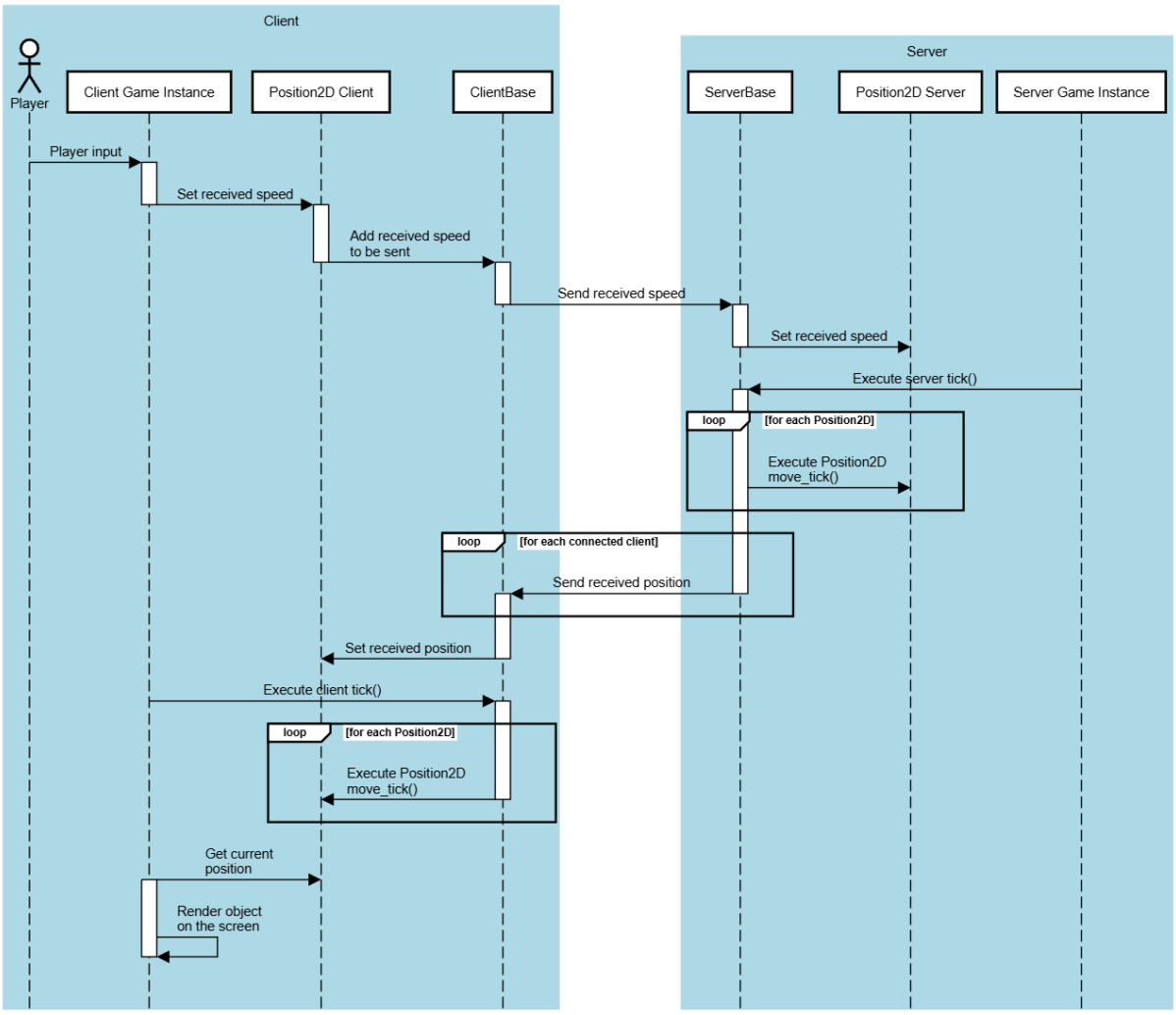

Figure 5.3 Sequence diagram of the `Position2D` update flow (source: own elaboration).

Listing 5.14 contains the constructor of the `Position2D` class, which gives insights into all functionalities provided by this component. It is possible to define boundaries for the x and y

coordinates to ensure that a given object will always stay in its designated area. Moreover, a simple automatic collision detection mechanism was introduced that can be used with objects of a rectangle shape. Usage of both those features is shown in the `_move_tick()` method depicted, where in case either an object tries to move out of the defined range, or a collision is identified, the movement is blocked. The last important parameter is the `max_speed`, which allows developers to set a limit for an acceptable received speed. It was introduced as a cheating prevention measure, as there is a risk a player might modify those values with malicious intent, therefore this check gives programmers the possibility to prevent unreasonably high values from affecting the gameplay.

In contrast to the `NetworkVariable` class, `Position2D` does not store a reference to either `ServerBase` or `ClientBase` class. However, both of those objects contain an instance of a dedicated container, `Position2dDictionary`, which stores references to all such members from existing `NetworkObjects`. As the movement updates are part of the corresponding `tick()` methods, such a design allows to access all instantiated positions, without the need to iterate through the entire `NetworkObjectDictionary`.

Listing 5.14 `Position2D` class constructor.

```
class Position2D:
    def __init__(self, init_x: float, init_y: float,
                 x_boundaries: (float, float),
                 y_boundaries: (float, float),
                 enable_auto_collisions: bool,
                 size: Optional[Tuple[float, float]] = None,
                 server_controlled: bool = False,
                 max_speed: Optional[float] = None):
        self.__x: float = init_x
        self.__y: float = init_y
        self.__size: Optional[Tuple[float, float]] = size
        self.__x_boundaries: (float, float) = x_boundaries
        self.__y_boundaries: (float, float) = y_boundaries
        self.__auto_collisions_enabled: bool = enable_auto_collisions
        self.__speed_x: float = 0.0
        self.__speed_y: float = 0.0
        self.__received_position: Optional[(float, float)] = None
        self.__received_speed: Optional[(float, float)] = None
        self.__base_type: Optional[BaseType] = None
        self.__owning_network_object: Optional[any] = None
        self.__server_controlled: bool = server_controlled
        self.__max_speed: Optional[float] = max_speed
```

An example of usage of the `Position2D` component can be found in listing 5.15, which presents a fragment of the implementation of a *Player* class in the *PyMP* example created with *pygame* module. This object serves as a controllable entity steered by a player. `Position2D` instance is a member of this class and its `speed_x` and `speed_y` properties are used in methods dedicated to movement in specific directions, and `position_x` and `position_y` values are utilized to get the object's current position.

Listing 5.15 Usage example of the `Position2D` component from the *PyMP* example created with *pygame* module.

```
class Player(NetworkObject):

    def __init__(self, size: (int, int), pos_x: int, pos_y: int, x_boundaries: (float, float),
y_boundaries: (float, float)):
        super().__init__()
        self.init_x = pos_x
        self.init_y = pos_y
```

```
        self.position2d = Position2D(pos_x, pos_y, x_boundaries, y_boundaries, True, size,
max_speed=5.0)
        self.color = (randint(0, 255), randint(0, 255), randint(0, 255))
        self.size: (int, int) = size

    def move_right(self, speed):
        self.position2d.speed_x = speed

    def move_left(self, speed):
        self.position2d.speed_x = -speed

    def move_up(self, speed):
        self.position2d.speed_y = -speed

    def move_down(self, speed):
        self.position2d.speed_y = speed

    @property
    def x(self):
        return self.position2d.position_x

    @property
    def y(self):
        return self.position2d.position_y
```

What has to be stated about the current `Position2D` implementation, is that it comes with limitations that require addressing during future development. Most notably, the current server-authoritative approach can lead to unsatisfying responsiveness on large networks with high latency. The possible approach to minimize this issue would be to introduce client-side prediction, which is discussed in section 5.3.2.

5.2.6 Events

Events are one of the low-level features of *PyMP*. This functionality gives developers the possibility to send arbitrary Python dictionaries that can be retrieved in a list. The flow of the process can be found in the sequence diagram presented in figure 5.4. This example shows events being sent from a client, however, in case they were to be created on the server side, the procedure would be identical. Firstly the events are generated in the game instance runtime and proper messages are added to a queue in `ClientBase` or `ServerBase` object. All created events are then sent to a declared recipient and are added to a buffer in the corresponding *Base* instance. Then a local game instance can fetch the received list. By doing so, it will obtain its copy, and the actual buffer will be cleared. To finalize the process, it is assumed that the game instance will iterate through the entire collection and will conduct specific actions based on the acquired content.

The engine offers the possibility to send such events from a client to a server, and from a server to a single or all connected players. In practice, this can be achieved using the `send_event()` method which is provided in both `ClientBase` and `ServerBase` classes. Additionally, thanks to the *base* reference inside of `NetworkObjects`, it is possible also to use this function inside defined Remote Procedure Calls. An example of usage of `send_event()` can be found in listing 5.16, which shows a method used to declare a winner in the developed introductory project with *pygame*. In this case, after the score of a given player exceeds five, all connected players will receive the dictionary passed as a parameter as an event.

Listing 5.16 Usage example of the `ServerBase.send_event()` method.

```
    def check_if_game_finished(self):
        if self.scoreboard.away_score_net_var.value == 5:
            self.server.send_event({"winner": "AWAY"}, MessageDestination.AllClients)
        elif self.scoreboard.home_score_net_var.value == 5:
            self.server.send_event({"winner": "HOME"}, MessageDestination.AllClients)
```

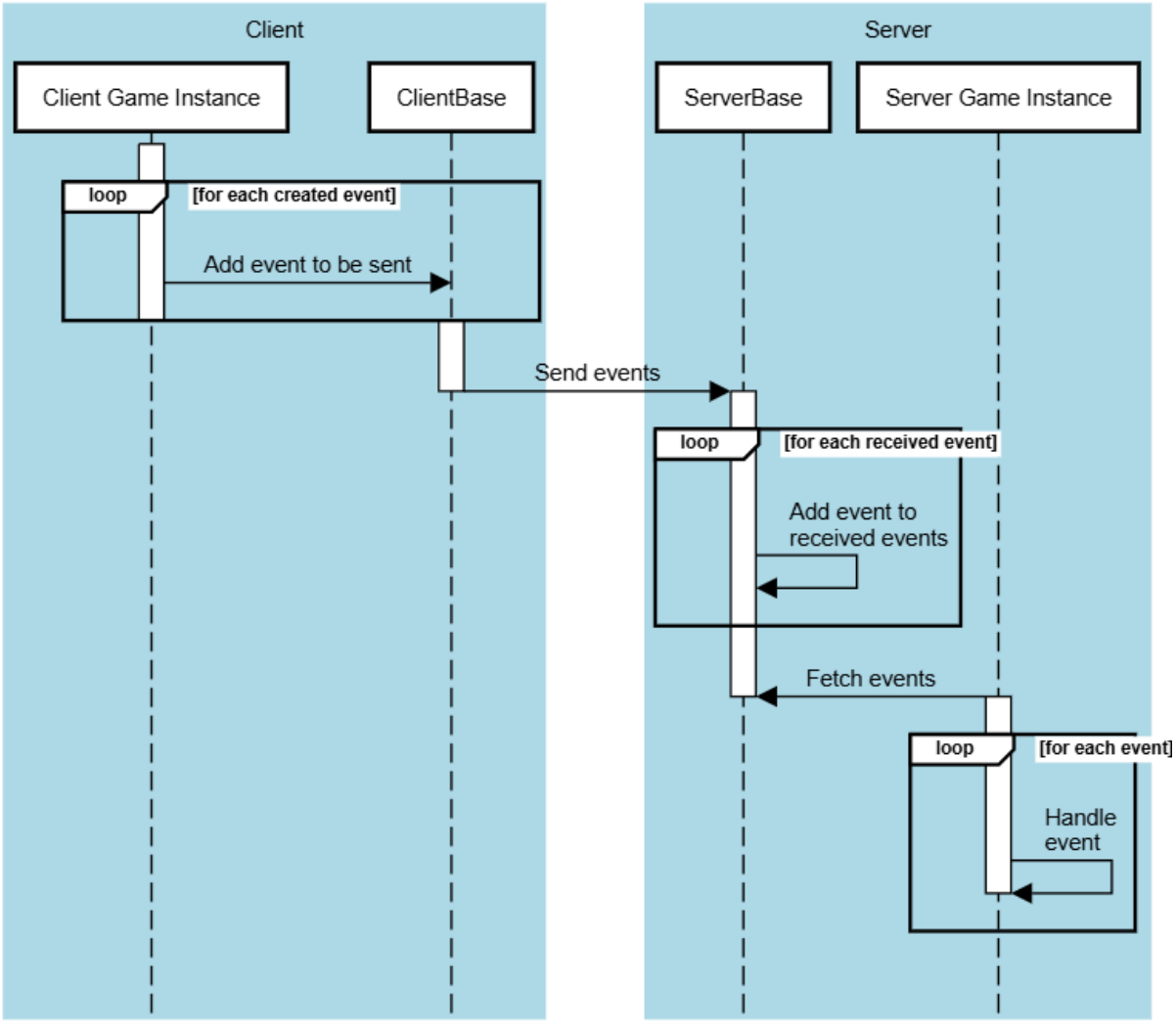

Figure 5.4 Event sending sequence diagram (source: own elaboration).

On the other hand, listing 5.17 presents how the event sent in listing 5.5 is received on the client side. Both the `ClientBase` and `ServerBase` classes contain a `received_events` property, which returns a Python list with all received events. The intended usage scenario of this functionality would be to iterate through the entire obtained collection during each game loop iteration and conduct specific actions based on the member values. As *PyMP* does not enforce any structure on used dictionaries to provide greater flexibility, it is up to the developer to decide how they will identify given events. In the case of the example presented in listing 5.17, all received events were checked if they contained an item with the key *"winner"*.

Listing 5.17 Usage example of *ClientBase.received_events* property.

```
def check_server_events(self):
    for event in self.client.received_events:
        if event.get("winner", None) is not None:
            self.winner = event["winner"]
            self.quit_button.show()
```

Even though the engine allows any dictionary to be used as an event, it has to be remembered that the *msgpack* library is used for serialization of those objects, and due to this fact there are some

limitations. Most notably as mentioned in section 4.3 this module cannot be used with arbitrary classes, therefore all the dictionaries should contain only standard data types.

5.2.7 Lobby

As was stated in section 3.2, the lobby is a vital functionality of online multiplayer video games, as it is a state before the required number of players can group before the actual gameplay begins. In *PyMP*, the game lobby is started by default after the `run()` methods of `ClientBase` and `ServerBase` objects are executed. It is active until the `ServerBase.start_game()` method is called, which sends a message to all clients to inform them that the gameplay should begin. An example of such an implementation is presented in listing 5.17, which contains the main function of the server implementation from the *PyMP* sample project developed with *pygame*. As can be seen, before the actual game loop another one exists, which is used for the purpose of the game lobby. In this state, among other things, it is possible to access the number of connected clients and a Boolean value stating if all players are ready to start. Those values can be used to create conditions for gameplay initiation, as shown in the example in listing 5.18. During the lifespan of the lobby, in each request, clients send a heartbeat message that contains:

- a field describing their readiness based on `ClientBase.is_ready()` property;
- an arbitrary Python dictionary that is intended to contain information describing a player, such as their nickname;

In response from the server, clients receive a snapshot of those values for each connected client, which can be used, for example, to display if a player is ready to start the game in the lobby user interface (UI), as it is presented in listing 5.19, showcasing a method for rendering the lobby UI from the *PyMP* sample project using *pygame*. The `ClientBase.lobby` property returns a dictionary with pairs of `client_id` and `ClientData` structures, which contain the `is_ready` value and custom player information. This can be utilized as shown in listing 5.19, to list all players in the lobby and mark their current readiness in the UI.

Listing 5.18 The main function of the server implementation from the *PyMP* sample project using *pygame*.

```
def main(self):
      self.server.run()
      while not self.server.game_started:  # lobby
          self.clock.tick(60)
          if self.server.number_of_connected_clients >= 2 and self.server.all_clients_ready:
              self.server.start_game()
      while True:
          if not self.setup_done:
              self.setup_game()
              self.setup_done = True
          self.clock.tick(60)
          if self.ball is None or self.scoreboard is None:
              self.ball = self.server.get_network_objects_by_class(Ball)[0]
              self.scoreboard = self.server.get_network_objects_by_class(Scoreboard)[0]
          self.ball.move_on_collision(self.server.get_network_objects_by_class(Player))
          self.server.tick()
          self.ball.slow_down()
          self.check_if_goal_scored()
          self.check_if_game_finished()
```

Listing 5.19 Function for rendering lobby UI in client's implementation in *PyMP* sample project using *pygame*.

```
    def blit_screen_lobby(self, time_delta):
```

```
        init_pos = (50, 50)
        self.screen.blit(self.empty_background_sprite, self.empty_background_sprite.get_rect())
        for client_id, client_data in self.client.lobby.items():
            text_surface = self.lobby_font.render(f'Player {client_id} {"(You)" if client_id ==
self.client.client_id else "       "}         [{'Ready' if client_data.is_ready else 'Not
ready'}]', False, (0, 0, 0))
            self.screen.blit(text_surface, init_pos)
            init_pos = init_pos[0], init_pos[1] + 60
        self.ui_manager.update(time_delta)
        self.ui_manager.draw_ui(self.screen)
        pygame.display.flip()
```

5.2.8 ClientStatus

In an ideal scenario, all connected clients keep communicating with the server throughout the entire game duration. In reality, however, connection might be lost due to, for example, network issues. Some players might also need to consciously resign due to various reasons. With this knowledge in mind, the *PyMP* engine automatically checks the state of each client and allows developers to access this information in the `ClientBase` and `ServerBase` instances.

Each player that has connected to the game serves has a status assigned, which is a value of the `ClientStatus` enum. All its possible instances can be found in a state diagram presented in figure 5.4. This chart also shows what status changes can occur in the system. There are three possible states players can be in:

- **`OK`:** The default `ClientStatus` value, which is set when a server receives the first request from a given client. In case a user does not decide to leave a game during the gameplay and there are no network problems that could cause connectivity issues, this value is maintained until the game is finished;
- **`DISCONNECTED`:** This value is set if a player consciously decides to leave a game. When it occurs, the local game instance can call the `ClientBase.disconnect()` method, which will send a message to inform the server about the decision;
- **`LOST_CONNECTION`:** This status is assigned automatically to a client in case the game server does not receive any request from them for 10 seconds, indicating potential technical problems;

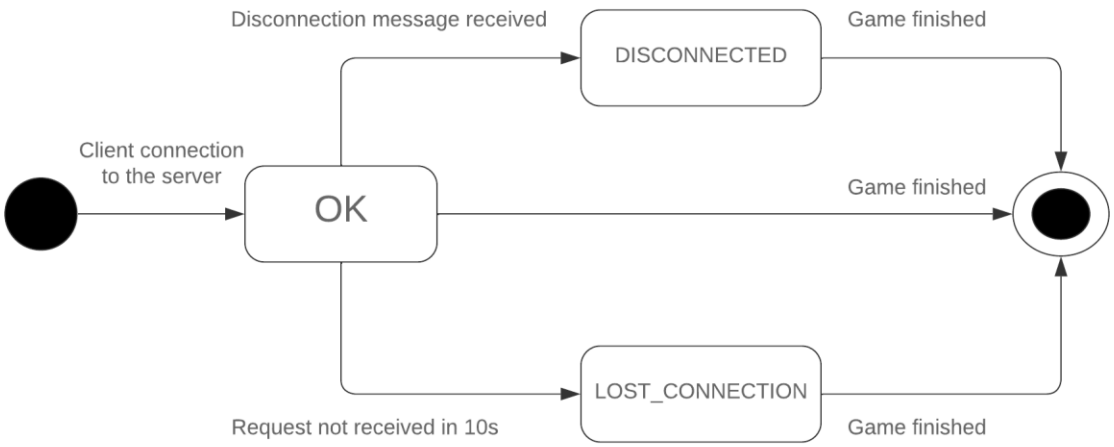

Figure 5.5 State diagram of `ClientStatus` in *PyMP* (source: own elaboration).

As can be seen in the diagram in figure 5.4, in the current implementation of *PyMP*, there is no possibility of reconnecting to a game, regardless if the communication was stopped intentionally or not. This is a limitation that requires addressing in the further development of the framework, which is discussed in section 5.3.2.

The status of all clients can be accessed thanks to the `client_statuses` property, which is available in both `ClientBase` and `ServerBase` classes and returns a dictionary with pairs of `client_id` and `ClientStatus` for each player. A simple example of the usage of this structure can be found in the method used to render in-game entities on the screen in client implementation in *PyMP* sample project using *pygame* presented in listing 5.20. There before each `Player` object is drawn on the surface, it is checked if its owner has an `OK` status in order to only show objects that belong to active players.

Listing 5.20 Method used to render objects on a screen in client game instance from *PyMP* example project using *pygame*.

```
    def blit_screen(self):
        self.screen.fill("white")
        self.screen.blit(self.field_sprite, self.field_sprite.get_rect())
        for network_object in self.client.get_network_objects_by_class(Player):
            player: Player = network_object
            if self.client.client_statuses[player.owner_id] == ClientStatus.OK:
                pygame.draw.rect(self.screen, player.color, pygame.Rect(player.x, player.y, 30,
30))
        for network_object in self.client.get_network_objects_by_class(Ball):
            ball: Ball = network_object
            pygame.draw.rect(self.screen, (0, 0, 0), pygame.Rect(ball.x, ball.y, 10, 10))
        self.blit_scoreboard()
        pygame.display.flip()
```

5.2.9 Integration with MongoDB database

The most unique feature available in the *PyMP* engine is the integration with the MongoDB database. This allows developers to create a connection to a database on a game server, resulting in frequent entries of game state snapshots and important events, such as a player joining a game. The feature can be enabled in the constructor of the `ServerBase` class by the `use_analytical_database` Boolean parameter. It is important to mention that the framework will not start a MongoDB instance on its own. Game programmers or server administrators interested in using this functionality are required to start it manually or using other tools. To set the connection in the engine it is required to set the IP address and the port of the database in dedicated fields in `ServerBase's` constructor.

Listing 5.21 Method for periodic database writes in `ServerBase` class.

```
    def __write_to_db(self):
        next_tick: Optional[datetime] = None
        while self.__server_running:
            self.tick_rate(1)
            now = datetime.now()
            if not next_tick or now > next_tick:
                next_tick = now + timedelta(seconds=self.__database_writes_interval_seconds)
                self.__database_handler.write_server_event_entries_from_queue()
                self.__database_handler.write_game_state_entry(
                                            self.__network_object_dict.get_snapshot(),
                                            self.get_game_duration())
```

If the MongoDB integration functionality is turned on, in the `ServerBase.run()` method in addition to the start of an asynchronous server, a connection to the database will occur, and a separate

thread handling periodic writes will be started. The method used for entry creation is presented in listing 5.21. It contains a loop active during the entire lifespan of the communication server where new database entries are being added in a frequency specified in the `ServerBase` constructor. On default, new writes will be done every 30 seconds.

The engine creates two types of *collections* in the MongoDB instance. The first one, called *server-events* contains entries regarding events such as a client connection to a server and is shown in Figure 5.6. The second one named *game-state*, consists of a snapshot of all existing `NetworkObjects` with all their `NetworkVariables` and `Position2D` members, as presented in Figure 5.7. In both cases, each *collection* name will include the date and the time the game has started. This allows us to associate the data with specific contests.

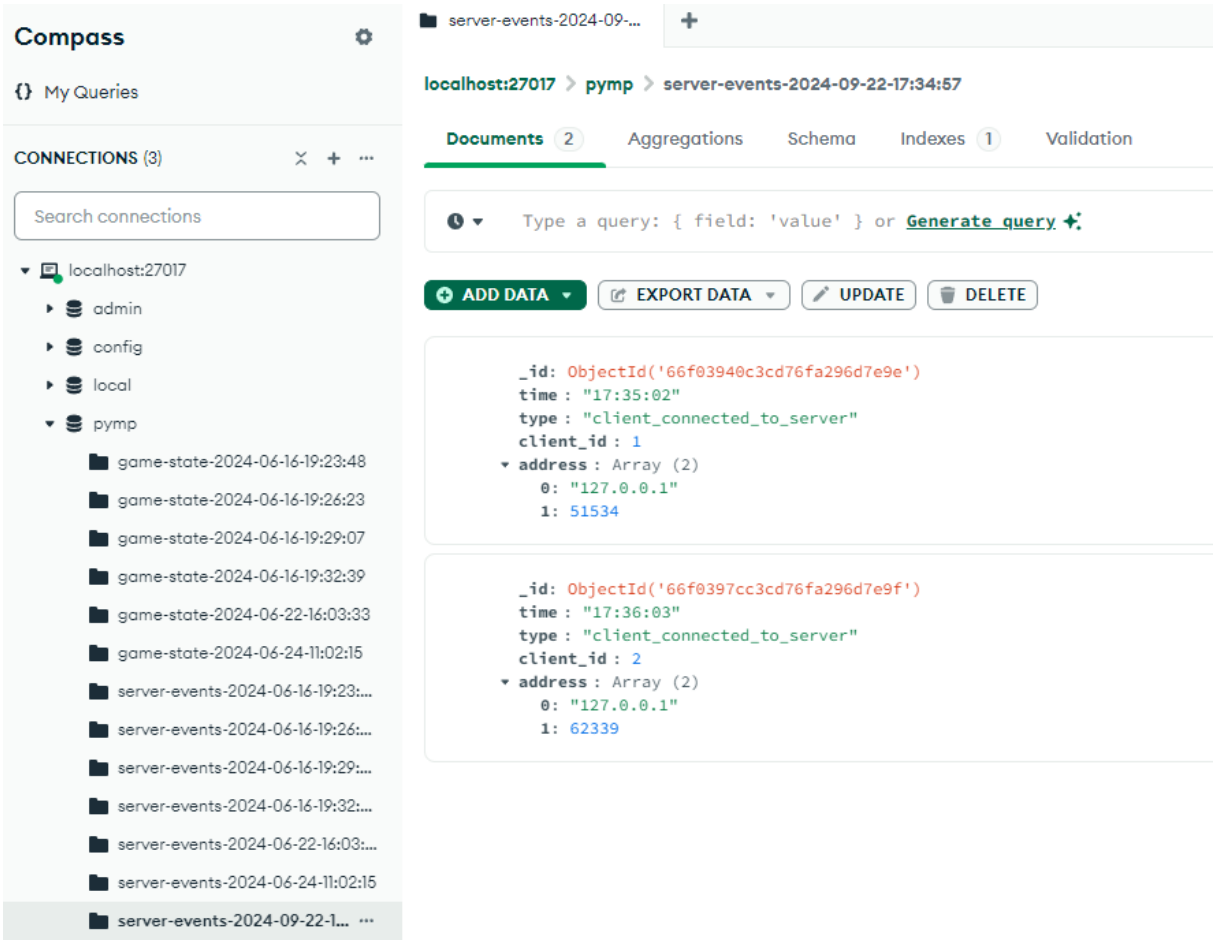

Figure 5.6 Screenshot of MongoDB Compass application showing an example of server-events collection.

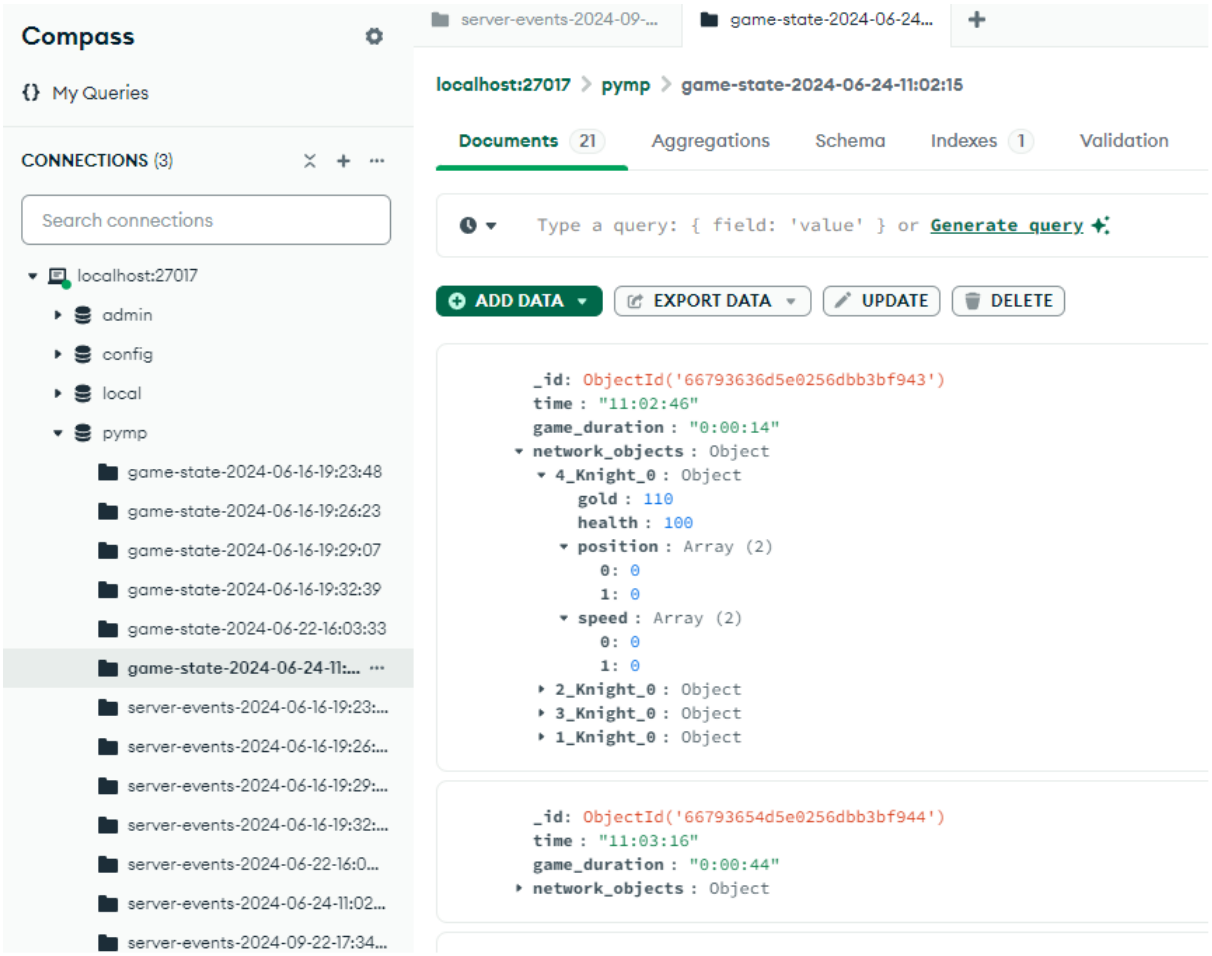

Figure 5.7 Screenshot of MongoDB Compass application showing an example of game-state collection.

5.3. PyMP engine discussion

This section contains a discussion regarding the performance of the implemented *PyMP* prototype and its strengths and areas of possible improvements, briefly outlining the plans for future development.

5.3.1 Prototype performance

To benchmark the performance of the *PyMP* engine tests similar to the ones described in section 3.4.3 were conducted. It was decided that the command-line sample project of the framework would be used for this purpose. In this case, the range of connected clients was chosen to be 4 to 32, each value being a next power of 2. Additionally, the experiments were repeated twice. Once using the User Datagram Protocol, and once using the Transmission Control Protocol. The gathered results can be found in table 5.1. What can be stated is that the performance of *PyMP* in the case of UDP tests is very similar to one measured for a created PoC. With substantial confidence, it can be stated that the engine should be able to handle at least 32 concurrent players with very low latency in UDP mode. What can be said about the usage of TCP on the other hand is that following the general belief, it performed significantly worse than the UDP counterpart. However, less than 5ms of response time for 32 concurrent players is assumed to be still sufficient for many video game use cases. It is especially true for genres like real-time strategy and virtual board games, where quick response times are not as

crucial as in more action-based ones. In conclusion, the performance test results look very promising. It has to be remembered that the example used in the experiments is relatively simple, yet regardless the expectation is that the engine should be able to handle at least 32 players at once with satisfying average response times.

Table 5.1 Performance test results of the PyMP command-line sample project using UDP and TCP (less is better).

Number of clients	Average response time [ms]	
	UDP	TCP
4	0.33255	0.59042
8	0.53130	1.21107
16	0.70766	2.96047
32	0.86609	4.24526

5.3.2 Strengths and areas for improvement

Acknowledged strengths of the implemented *PyMP* engine prototype:

- **Complete synchronized game state:** The framework offers the most complete synchronized game state system in comparison to analysed solutions that currently exist on the market. None of the analysed libraries gives the possibility to create shared objects representing in-game entities that are maintained on both the client and the server side, have assigned ownership, and are easily accessible by the developer.
- **`NetworkVariable` and RPC functionalities:** The `NetworkVariable` and `rpc` decorator functionalities of *PyMP* neatly coexist with the framework's emphasis on the object-oriented programming paradigm. They allow developers to encapsulate both client-sided and server-sided functionalities of in-game entities in a single class implementation, significantly reducing the required amount of work.
- **Built-in in-game entity movement and collision handling:** The *PyMP* engine offers the `Position2D` feature for the movement of in-game entities. It is independent of the module used for game development, which gives developers freedom of choice. Additionally, it also comes with simple rectangle collision handling.
- **Built-in lobby functionality:** The *PyMP* engine comes with an integrated lobby functionality. This relieves the developers of the need for manual implementation of this important feature.
- **An integration with MongoDB:** The integration with the MongoDB database is a unique feature of *PyMP*, which can be utilized after a game's release to discover player behaviour patterns and to detect cheating attempts, as well as during development as a debugging tool.

Identified areas of possible improvements of the implemented *PyMP* engine prototype:

- **Client-side prediction and server reconciliation of player position:** As was brought up in section 5.2.5, there are a few limitations of the `Position2D` component that require further addressing. Most notably due to the server-authoritative design, the actual movement will happen after it is processed and received from a server. In the case of small networks such as LAN, it does not impose significant problems. However, on larger ones with higher latency such as WAN, the gameplay responsiveness will be degraded, which is problematic for action-based games. As mentioned in [58], this issue could be mitigated by adding client-side prediction and server reconciliation mechanisms. The new position of objects can be assumed

based on previous values and before an actual update arrives from a server, objects can be rendered based on calculated guesses. Having such a mechanism would ensure that the communication latency does not degrade the player experience. What has to be kept in mind though, is that it could lead to potential desynchronization. Therefore the `Position2D` component should correct the predicted values based on the data from the server.

- **Integration with *pygame* module:** The design of *PyMP* does not enforce the usage of any particular framework for the development of video games, which gives programmers flexibility and freedom of choice. However, there are use cases where integration with a module like *pygame* could be very beneficial. For example, in `Position2D` better solution would be to send the player's input instead of speed and a proper implementation of such functionality needs to be done with a particular library in mind.
- **Client reconnections:** A crucial limitation of the current *PyMP* implementation is that there is no possibility of reconnecting to a server after a connection has been lost. This requires further addressing, also giving developers ways to manage the state of disconnected players. Programmers should be able to define the conditions a player needs to fulfil to be allowed to restore connection and should be able to remove or modify objects owned by a disconnected player.
- **Network reliability:** UDP was chosen as the default communication protocol due to superior performance, yet it has to be remembered that its lack of reliability can cause issues. The best solution would be to handle problems such as lost packets in *PyMP*, as it is done for example in the *MpGameServer* library.
- **Security enhancements:** Many additional functionalities can be added to *PyMP* to improve the security of the solution. Those include communication encryption and advanced client authorization. This would ensure that the message exchanged between clients and the server would be safe from, for example, man-in-the-middle attacks and that only welcomed players would be able to connect to a game.

5.3.3 Plans for future development

The objective is to publish *PyMP* as an open-source project. Considering that, the first further development step would be a release in the Python Package Index (PyPI). By doing so, it would become publicly available to developers, who could provide their feedback and observations. Ideally, a build pipeline can be set up on the GitHub repository, which would automatize the publication process. When it is accomplished, the plan is to prioritize all the improvements and new functionalities discussed in section 5.3.2. Doing so will enable the creation of a roadmap of new features that will be consistently added.

6. Conclusions

Development of online multiplayer video games is not a simple task, as it requires expertise in computer networking and practical experience to deal with challenges unique to distributed systems. There were many attempts to create Python frameworks to alleviate programmers from many of the related pains, a relevant subset of which was analysed as shown in the 2nd chapter of the work. Considering their identified strengths and weaknesses, as well as the conducted research and practical experiences an idea for a new solution was painted in words in the 3rd chapter. Crucial challenges the engine needs to address were outlined in section 3.1 and a list of most important required functionalities was provided in section 3.2. To properly tackle the problem, the topic of concurrency in Python was explored. Multiprocessing and asynchronous programming were selected as potential alternatives to traditional threading, which is limited by the *Global Interpreter Lock*. Based on created Proofs of Concept, their performance was assessed and asynchrony was chosen as a technique most suitable for the task at hand. Most importantly, the measured efficiency was promising and gave confidence that the creation of a performant engine is entirely feasible.

The fifth chapter of the work contains a detailed description of the implemented *PyMP* engine prototype. All requirements stated in section 3.2 were successfully fulfilled. Additionally, two sample projects were delivered to provide a convenient entry point to the framework. It has its unique strengths, such as a complete synchronized game state based on shared `NetworkObjects` that differentiates it from other available solutions. Features such as the `NetworkVariable`, and the `rpc` method decorator greatly reduce the required amount of work, simplifying the entire game creation process. Great effort was put into abstracting complex functionalities and providing a simple interface. This can enable novice programmers to develop their dream projects lowering the required entry level. There are still many things that need to be done for *PyMP* to become a complete solution, as mentioned in section 5.2.3. However, with proper support, the engine has the potential to become an exceptionally useful tool for creating online multiplayer games in Python.

Bibliography

[1]. A Brief History of Mirror https://mirror-networking.gitbook.io/docs/trivia/a-history-of-mirror (date of access 09.11.2024)

[2]. Democratizing Multiplayer Development - Unite Europe 2015 https://www.youtube.com/watch?v=gZbbYXzyXKk&ab_channel=Unity (date of access 09.11.2024)

[3]. Introduction to GameObjects - Unity Manual https://docs.unity3d.com/Manual/GameObjects.html (date of access 09.11.2024)

[4]. Introduction to Components – Unity Manual https://docs.unity3d.com/Manual/Components.html (date of access 09.11.2024)

[5]. Network Manager - Mirror documentation https://mirror-networking.gitbook.io/docs/manual/components/network-manager (date of access 09.11.2024)

[6]. Network Identity - Mirror documentation https://mirror-networking.gitbook.io/docs/manual/components/network-identity (date of access 09.11.2024)

[7]. Multiplayer strategy-economics game implementation – Adam Furche, Konrad Trzmielewski, December 2022 (author's Bachelor thesis)

[8]. MonoBehaviour - Unity Manual https://docs.unity3d.com/Manual/class-MonoBehaviour.html (date of access 09.11.2024)

[9]. Network Behaviour - Mirror documentation https://mirror-networking.gitbook.io/docs/manual/components/networkbehaviour (date of access 09.11.2024)

[10]. SyncVars - Mirror documentation https://mirror-networking.gitbook.io/docs/manual/guides/synchronization/syncvars (date of access 10.11.2024)

[11]. Commands - Mirror documentation https://mirror-networking.gitbook.io/docs/manual/components/networkbehaviour#commands (date of access 10.11.2024)

[12]. Client RPC Calls - Mirror documentation https://mirror-networking.gitbook.io/docs/manual/components/networkbehaviour#client-rpc-calls (date of access 10.11.2024)

[13]. PodSixNet GitHub repository - https://github.com/chr15m/PodSixNet (date of access 10.11.2024)

[14]. asyncore - Python documentation https://docs.python.org/3.11/library/asyncore.html (date of access 10.11.2024)

[15]. PyGaSe GitHub repository - https://github.com/sbischoff-ai/pygase (date of access 10.11.2024)

[16]. PyGase API Reference - https://sbischoff-ai.github.io/pygase/ (date of access 18.12.2024)

[17]. Context Manager Types – Python documentation - https://docs.python.org/3/library/stdtypes.html#context-manager-types (date of access 27.11.2024)

[18]. Simple-Game-Server GitHub repository - https://github.com/Ganapati/Simple-Game-Server (date of access 10.11.2024)

[19]. MpGameServer GitHub repository - https://github.com/nsetzer/mpgameserver (date of access 10.11.2024)

[20]. About – pygame wiki - https://www.pygame.org/wiki/about (date of access 18.12.2024)

[21]. Panda 3D Manual - https://docs.panda3d.org/1.10/python/index (date of access 22.12.2024)

[22]. Networking – Panda 3D manual - https://docs.panda3d.org/1.10/python/programming/networking/index (date of access 10.11.2024)

[23]. Peer-to-Peer vs. Dedicated Server – A Comparison - Tobias Mildenberger https://contabo.com/blog/peer-to-peer-vs-dedicated-servers/ (date of access 01.12.2024)

[24]. Fast-Paced Multiplayer (Part 1): Client-Server Game Architecture – Gabrel Gambetta https://www.gabrielgambetta.com/client-server-game-architecture.html (date of access 01.12.2024)

[25]. Remote Procedure Call (RPC) – Alexander S. Gillis https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC (date of access 25.11.2024)

[26]. TCP vs. UDP: Understanding 10 Key Differences – Chiradeep BasuMallick https://www.spiceworks.com/tech/networking/articles/tcp-vs-udp/ (date of access 04.12.2024)

[27]. Tick and update rates – Unity Documentation - https://docs-multiplayer.unity3d.com/netcode/1.9.1/learn/ticks-and-update-rates/ (date of access 22.12.2024)

[28]. Mastering Concurrency in Python - Quan Nguyen, November 2018, ISBN 978-1-78934-2

[29]. PyCon 2015 – Python's Infamous GIL by Larry Hastings https://www.youtube.com/watch?v=KVKufdTphKs&list=PLP05cUdxR3KsS3yWl5LRiko2lRAp1XPUd&ab_channel=AlphaVideoIreland (date of access 01.12.2024)

[30]. Releasing GIL and mixing threads from C and Python - https://www.geeksforgeeks.org/releasing-gil-and-mixing-threads-from-c-and-python/ (date of access 22.12.2024)

[31]. Free-threaded CPython – Python documentation https://docs.python.org/3/whatsnew/3.13.html#free-threaded-cpython (date of access 01.12.2024)

[32]. multiprocessing.sharerd_memory – Shared memory for direct access across processes https://docs.python.org/3/library/multiprocessing.shared_memory.html (date of access 04.12.2024)

[33]. multiprocessing.shared_memory.ShareableList – Python documentation https://docs.python.org/3/library/multiprocessing.shared_memory.html#multiprocessing.shared_memory.ShareableList (date of access 04.12.2024)

[34]. An Introduction to Asynchronous Programming in Python - https://medium.com/velotio-perspectives/an-introduction-to-asynchronous-programming-in-python-af0189a88bbb (date of access 09.12.2024)

[35]. asyncudp documentation - https://asyncudp.readthedocs.io/en/latest/ (date of access 14.12.2024)

[36]. asyncio.run – Python documentation https://docs.python.org/3/library/asyncio-runner.html#asyncio.run (date of access 09.12.2024)

[37]. Network Latency and Throughput - https://www.hostafrica.com/blog/networks/network-latency-and-throughput (date of access 10.12.2024)

[38]. Foreword to "Programming Python" (1st ed.) – Guido van Rossum https://legacy.python.org/doc/essays/foreword/ (date of access 24.11.2024)

[39]. The Top Programming Languages 2024 – Stephen Cass https://spectrum.ieee.org/top-programming-languages-2024 (date of access 24.11.2024)

[40]. Yawnoc - Steam page - https://store.steampowered.com/app/2824730/Yawnoc/ (date of access 14.12.2024)

[41]. DaFluffyPotato YouTube channel - https://www.youtube.com/@DaFluffyPotato (date of access 14.12.2024)

[42]. Streams – Python documentation https://docs.python.org/3/library/asyncio-stream.html (date of access 14.12.2024)

[43]. pickle – Python object serialization https://docs.python.org/3/library/pickle.html (date of access 11.12.2024)

[44]. Don't Pickle Your Data – Ben Frederickson https://www.benfrederickson.com/dont-pickle-your-data/ (date of access 11.12.2024)

[45]. Python Pickle Risks and Safer Serialization Alternatives – Alyse Osbourne https://arjancodes.com/blog/python-pickle-module-security-risks-and-safer-alternatives/ (date of access 14.12.2024)

[46]. What Is MongoDB? An Expert Guide – Jefferey Erickson https://www.oracle.com/pl/database/mongodb/ (date of access 15.12.2024)

[47]. What is MongoDB Compass? https://www.mongodb.com/docs/compass/current/#std-label-compass-index (date of access 18.12.2024)

[48]. About SDL - https://www.libsdl.org/ (date of access 18.12.2024)

[49]. pygame documentation - https://www.pygame.org/docs/ (date of access 18.12.2024)

[50]. Top Python Game Engines For Developers – Balazs Refi https://bluebirdinternational.com/python-game-engines/ (date of access 21.12.2024)

[51]. What is Git? https://www.atlassian.com/git/tutorials/what-is-git (date of access 21.12.2024)

[52]. GitHub – Ben Lutkevich https://www.techtarget.com/searchitoperations/definition/GitHub (date of access 21.12.2024)

[53]. What is PyCharm? Features, Advantages, and Disadvantages – Simran Kaur Aurora https://hackr.io/blog/what-is-pycharm (date of access 17.12.2024)

[54]. pygame.time.Clock - pygame documentation https://www.pygame.org/docs/ref/time.html#pygame.time.Clock (date of access 02.01.2025)

[55]. What is Reflective Programming – Kelechi Onyekwere https://khelechy.medium.com/what-exactly-is-reflective-programming-a-practical-example-a5a1015bdd3f (date of access 04.01.2025)

[56]. getatrr() – Python documentation https://khelechy.medium.com/what-exactly-is-reflective-programming-a-practical-example-a5a1015bdd3f (date of access 04.01.2025).

[57]. PEP 726 – Module __setattr__ and __delattr__ - https://peps.python.org/pep-0726/ (date of access 05.01.2025)

[58]. Fast-Paced Multiplayer (Part II): Client-Side Prediction and Server Reconciliation – Gabriel Gambetta https://www.gabrielgambetta.com/client-side-prediction-server-reconciliation.html (date of access 07.01.2025)

Appendix

Appendix A: List of used terminology and abbreviations

2D – two-dimensional

3D – three-dimensional

API – Application Programming Interface

DVCS – Distributed Version Control System

GIL – Global Interpreter Lock

GUI – Graphical User Interface

JSON – JavaScript Object Notation

LAN – Local Area Network

P2P – Peer-to-Peer

PEP – Python Enhancement Proposal

PoC – Proof of Concept

PyPI – Python Package Index

RPC – Remote Procedure Call

SDL - Simple DirectMedia Layer

TCP – Transmission Control Protocol

Tick rate - frequency of server game state updates, measured in Hertz

UDP – User Datagram Protocol

UI – user interface

Update rate - frequency of client sending and receiving data from a server, measured in Hertz

WAN – Wide Area Network

VCS – Version Control System

Appendix B: List of Figures

Appendix C: List of Listings

Appendix D: List of Tables