COMPUTER NETWORKS

Online Pong

Authors
Matias Barandiaran, Iason Iacovides, Jiashuo Hu
Lancaster University Leipzig

March 12, 2024

Contents

1 Revision History

Version	Name	Reasons for Changes	Date
1.0	FAT Server	Easy to deploy as most of the work is done by the server.	13/02/24
2.0	FAT Client	Improved performance by utilizing local resources.	01/03/24

2 Approved by

Name	Status	Department	Date
Aakash Ahmad	Approved	Computer Science	6/02/24

3 Introduction

This report focuses on the implementation of a simple and interesting two-dimensional game, Pong, designed to be played by individuals connected to the same network. In this report we will outline the objectives of our project, provide an overview of the methodology which we used and highlight problems which we faced and how we overcame them. For a detailed view on the code structure, please refer to our GitHub Repository.

3.1 System Purpose

The purpose of this project was to implement a simple online game using Python as our programming language. In this system we aimed to have two users connected to a centralised server. This will be accomplished by forwarding data via socket programming. It is important to clarify that there are no intentions on releasing the final product for consumption, but this is rather a proof of concept for functional network technologies.

3.2 Challenges

Pong is a two-dimensional sports game that simulates the dynamics of table tennis. The player controls an in-game paddle by moving it vertically across the left or right side of the screen. The main objective is to strike a ball within the screen's confines, preventing it from breaching each player's side. Every time the ball hits one's side, the opposing player gains a point. Competing against each other, players strive to attain the highest score possible [3].

Given that Pong is naturally a real-time (RT) game, fast data forwarding and processing is imperative to maintain an optimal player experience. This is not the case for turn-based games, like Chess, where data reliability is of the highest priority. This, of course, raises valid questions regarding the pertinent type of protocol for the project.

A good starting point is to look at UDP which, despite its unreliability, promises results of low-latency communication. However, this connectionless protocol introduces further problems like risks of network overflooding, data loss and poor congestion control.

In the beginning, every game was networked peer-to-peer (p2p), with "each computer exchanging information with each other in a fully connected mesh topology" [1]. Although

p2p may fit various RT games, history shows that this architecture carries various limitations. The most important drawback lies on the absence of a central point of authority, leading to difficulties in managing game sessions.

3.3 Solution Overview

To begin with, it was decided to opt for a client-server model. By centralizing authority on the server side, one can assure objective game session managing while implementing efficient data transmission protocols. Moreover, because of the unreliable nature of UDP, we chose to implement our own version of forward error correction (FEC). FEC algorithms will be deployed to detect and correct data transmission errors due to bit flipping, enhancing data reliability. On the same note, congestion control mechanisms will be implemented to regulate data flow and mitigate network congestion. For a more detailed explanation on how this was achieved, please refer to Section 4. A use case diagram was constructed outlining our solution plan:

Figure 1: Use Case Diagram

4 Network System Design

4.1 Functional Specifications

As required, three central network concepts were implemented: UDP, congestion control and error correction. Due to the nature of the application at hand, UDP was our protocol of choice. This can be observed in the configuration of both client and server sockets with the

parameter *SOCK_DGRAM* (refer to Appendix A.3). Opposed to TCP where a connection or handshake is established before any data exchange, UDP simply sends datagrams across the medium, which are then recovered at the respective side by specifying the buffer size (which was defined to be 1024 bytes). As explained above, because of this protocol's unreliability, FEC was implemented following a Reed-Solomon codes approach. Reed-Solomon codes is a non-cyclic technique particularly useful for burst-error correction [4]. Instead of designing a packet re-transmission procedure from scratch, it was decided to introduce redundant data on every packet to assess any instances of flipped bits at the receiving side. UDP was chosen because of its advantages in applications demanding responsiveness and low latency. While packet re-transmission may improve our program's reliability, this would defeat the point of using UDP in the first place. That being said, FEC is a reasonable attempt at striking a balance between speed and reliability.

The initial approach was to encode string data to send screen positions in CSV notation across the network. However, due to the lack of extensibility of sending strings only, the *pickle* library was used to serialize and de-serialize Python objects for their transmission. This would allow one to send information like color, direction, position, etc. Following a hybrid object oriented structure, Ball and Paddle classes were created. This meant that one could retrieve all the object's attributes at either side, facilitating interpolation and overall game state consistency.

Finally, because UDP lacks measures of congestion control, a procedure similar to that used in AIMD was deployed. This procedure is not network assisted, meaning that no data related to the network's current congestion status is sent in the packets. It was achieved by building a first come first served (FCFS) packet queue that would cap how many packets are sent per second (refer to Appendix A.1). A variable called *MAX_TRANS_RATE* was defined stating this value. Moreover, a list storing the last 100 transmission round trip times (RTT) was created. Every time a new packet was received, its RTT would get compared to the average of the last 100 values, if said measurement surpassed the average by a certain threshold, an extra delay would get applied for the next transmission, preventing the network from being over flooded.

4.2 Network Components

The program is divided into two central components, a client and a server. As stated in the Revision History section (refer to Section 1), it was decided to implement a FAT Client [1] architecture. The main reason for the selection of this approach has to do with the fact that FAT or Thick clients do not need to use any server computing resources, which results particularly useful when the server has slow network speeds or limited storage capacity. Some disadvantages that this architecture carries relate to state consistency across all clients, specially in game applications. These requires some extra computation steps like data interpolation.

[1] Software or application which does all its data processing, resource managements and performing of operations on the client side, without depending on the main server [2].

4.2.1 Client

This is the component responsible for the rendering and displaying of the game to the user. This was achieved via the *Pygame* library for game development due to their off-the-shelf drawing functions. Furthermore, working with a FAT client meant that all game logic would be handled in this side of the architecture, this included input and collision handling, updating paddle and ball position and score calculation.

Each client would be in charge of controlling the behaviour of one paddle, either left or right depending on the sequence in which they connected to the server. It is important to mention that "connected" is not meant in a TCP context, but rather each client would send a special reserved key word to the server notifying that they would like to start a game session. This would trigger the addition of their address into the address table mapped within the server.

The clients would then send the serialized paddle of their "player 1" (according to their own eyes) once every frame, while receiving the opponent's paddle from the server. This would allow for each client to display an updated version of the paddle positions with the lowest latency or delay possible. Now, due to the points raised above, ball interpolation had to be carried out. Due to inconsistencies in floating point calculations, the balls in both clients would eventually get out of sync. This was solved by taking the average of both balls' positions and exchanging their direction (since these are the discrete values 1 and -1).

To aid with network behaviour, the clients would create a *Network* object, which would handle aspects like FEC and congestion control. It is important to mention that packet queuing was only implemented on the client side, since these are the ones constantly sending data (around 180 times per second). Whereas the server did not require it, as it would only forward any appropriate information after receiving clients' instructions.

4.2.2 Server

The server would essentially act as a communication bridge between both clients. Two address tables (dictionary data structures) mapped each client's address with their respective attributes, paddles and balls. Moreover, a centralized score tuple was deployed which would get constatly updated by each client, facilitating interpolation. Furthermore, every time data was received, the data processor would check for special keywords like "end", "init_paddles", etc. for specific game session operations. Finally, Reed-Solomon correction was also performed at this component, preventing errors from propagating through the channel.

4.3 System Architecture

To showcase both version of our implementation, the initial (Figure 2) and final (Figure 3) deployment diagrams are presented below.

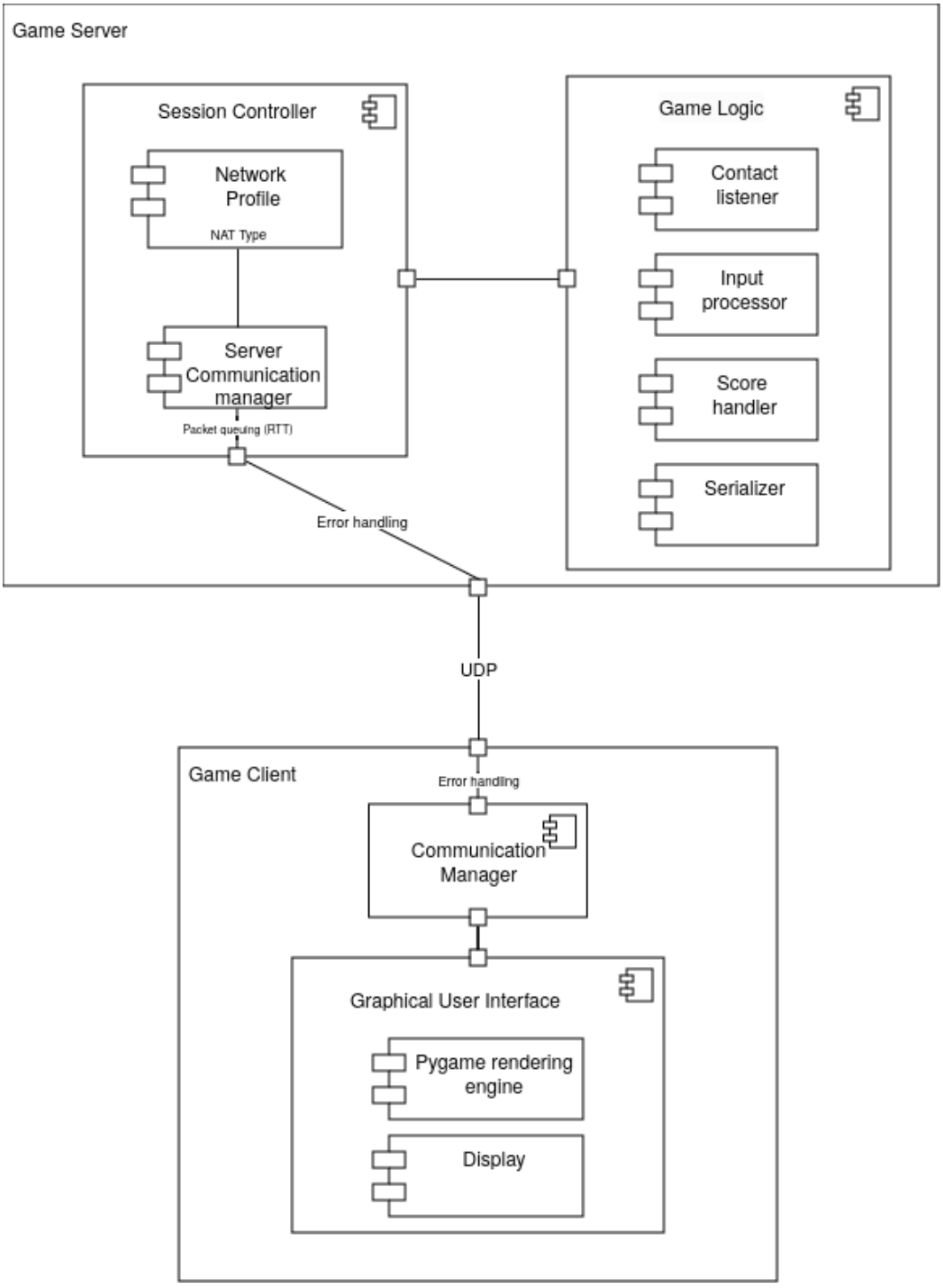

Figure 2: Initial Deployment Diagram

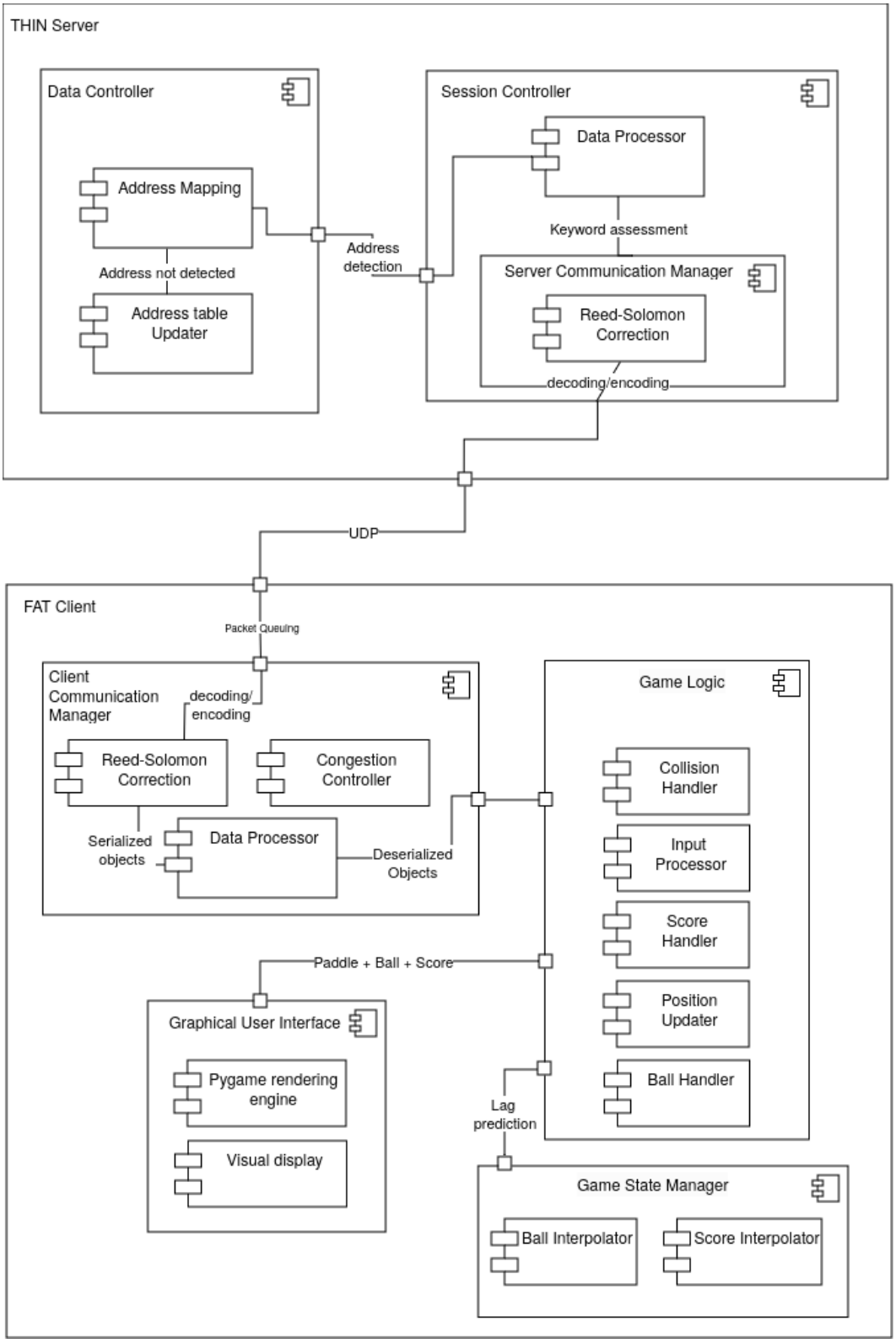

Figure 3: Final Deployment Diagram

5 System Testing

5.1 Unit Test Cases

- **Designed By:** Jiashuo Hu
- **Executed By:** Jiashuo Hu
- **Design Date:** 28/02/2024
- **Execution Date:** 03/03/2024
- **Release Version:** 2.0

Refer to Table 2 for test case information.

Test Case #	Test Title	Test Summary	Test Steps	Test Data	Expected Result	Status (Pass/-Fail)	Notes (if any)
1	test_reed_solo	Testing if Reed Solomon (RS) encoding and decoding works as indented.	Serialize the input data. RS encode the input data and use the result to receive the RS decoded data.	b'Test Data'	"Test Data"	Pass	None
2	test_interpolation	This test checks whether the interpolation of ball position and direction works correctly.	Create a dummy network instance, initialize a ball object, call the interpolate function with ball and network instances.	Ball object with x=100, y=100, direction=[1, -1] and mock received data as a Ball with x=200, y=150, direction=[0, 1]	The ball's x position should be updated to 150. The ball's y position should be updated to 125. The direction should be set to [0, 1]	pass	None
3	test_reset_position	Ensure positions are properly reset.	Intialize ball, change position attributes, check if ball's current x and y coordinates match the initial coordinates.	ball.initx=100, ball.inity=100, newx=50, newy=50	currx=100, curry=100	Pass	None

Table 2: Unit Test Cases

5.2 Stress Test Cases

- **Designed By:** Matias Barandiaran
- **Executed By:** Matias Barandiaran
- **Design Date:** 29/02/2024
- **Execution Date:** 01/03/2024
- **Release Version:** 2.0

Refer to Table 4 for test case information[2].

[2]Throughput expected value: Assuming 15 words per object, 360 objects sent each second distributed over 2 clients during game execution (180 = 3 objects * 60 FPS per client), on a 64 bit system.

Test Case #	Test Title	Test Summary	Test Data	Breakout Point	Safe Point	Status (Pass/Fail)	Notes (if any)
1	Noise Level	Measuring Reed Solomon correction after varying number of flipped bits.	"noisy text" converted into byte array.	6 bits	5 bits	Pass	None
2	Response Time	Measuring server response time after varying packet transmission delay (PTD).	Response time kept constant at around 5e-4 s.	None	1e-4 s	Pass	No breakout point due to congestion control procedures.
3	CPU Usage	Recording CPU usage (%) after varying PTD.	Spikes of 100% utilization detected at breakout point.	1e-6	1e-5	Pass	None
4	Memory Usage	Recording memory usage (%) after varying PTD.	Utilization kept constant at around 40%	None	All	Pass	Server never collection large amounts of memory.
5	Throughput	Recording program's throughput after varying PTD.	Expected at least 0.3 Mb/s	<1e-23	1e-23	Pass	None

Table 4: Stress Test Cases

References

[1] Gaffer On Games. Client server connection. `https://www.gafferongames.com/post/client_server_connection/`, Sep 2016.

[2] P Lowber. Thin client vs. fat client tco. *research note, Gartner*, 2001.

[3] Henry Lowood. Videogames in computer space: The complex history of pong. *IEEE Annals of the History of Computing*, 31(3):5–19, 2009. `doi:10.1109/MAHC.2009.53`.

[4] Bernard Sklar. Reed-solomon codes. *Downloaded from URL http://www. informit. com/content/images/art. sub.–sklar7. sub.–reed-solomo-n/elementLinks/art. sub.–sklar7. sub.–reed-solomon. pdf*, pages 1–33, 2001.

Appendices

A Code

A.1 Client Communication Manger Code

```
import random
import time
from socket import *
from queue import Queue
from network.reed_solomon import ReedSolomon
from helpers.constants import *
import pickle

# Helper class to encapsulate procedures related to network connection

class Network():

    def __init__(self, port, ip, bufsize):
        self.client = socket(AF_INET, SOCK_DGRAM)
        self.port = port
        self.ip = ip
        self.bufsize = bufsize
        self.address = (ip, port)

        self.corr = ReedSolomon(ECC)  # Forward error correction

        self.packet_queue = Queue()  # Packet queue for outgoing packets

        self.last_sent_time = time.time()
        self.last_received_time = None

        self.rtt_history = []

        self.delay = 0

    def enqueue_packet(self, data):
        self.packet_queue.put(pickle.dumps(data))

    def dequeue_packet(self):
        return self.packet_queue.get()

    def send(self, noise=False, noise_level=0):

        timed_delay = 0

        # Looping until we have sent every packet in the queue
        while not self.packet_queue.empty():

            # Calculating current time
            current_time = time.time()

            if self.delay != 0:
                timed_delay = current_time + self.delay
                self.delay = 0
```

```
50
51          if current_time < timed_delay:
52              continue
53
54          # Comparing it with the last time we sent a packet
55          time_since_last_sent = current_time - self.last_sent_time
56
57          # Preventing network from overflooding
58          if time_since_last_sent >= 1 / MAX_TRANS_RATE:
59              data = self.dequeue_packet()
60              encoded_data = self.corr.reed_solo_encode(data)
61              if noise:
62                  encoded_data = self.inject_noise(encoded_data,
    noise_level)
63              self.client.sendto(encoded_data, self.address)
64              self.last_sent_time = current_time
65
66  def inject_noise(self, data, noise_level):
67      num_errors = noise_level
68      for _ in range(num_errors):
69          index = random.randint(0, len(data) - 1)
70          data[index] ^= 1  # Flip a bit
71      return data
72
73  def receive(self):
74
75      data, _ = self.client.recvfrom(self.bufsize)
76
77      self.last_received_time = time.time()   # Updating received time
78
79      decoded_data = self.corr.reed_solo_decode(data)
80
81      return pickle.loads(decoded_data)
82
83  def assess(self):
84
85      rtt = self.getRTT()
86
87      if rtt is not None:
88
89          # Assesing only last 50 rtts
90          if len(self.rtt_history) > 100:
91              avg_rtt = sum(self.rtt_history) / len(self.rtt_history)
92              threshold = avg_rtt * 1e4  # Testing if the new RTT is 1e4
    times higher than the previous average
93              self.rtt_history.pop(0)  # Remove the oldest RTT from the
    history
94
95              if rtt > threshold:
96                  print("RTT exceeded threshold:", rtt)
97                  self.delay = 5e-6
98
99          self.rtt_history.append(rtt)
100
101 def getRTT(self):
102     if self.last_received_time is not None:
103         return self.last_received_time - self.last_sent_time
```

```
        else:
            return None


    def close(self):
        self.client.close()
```

Listing 1: network.py

A.2 FEC Code

```
from reedsolo import RSCodec

# Helper class for handling forward error correction(FEC)
# Uses reed solomon codes
class ReedSolomon():

    def __init__(self, ecc):
        self.rsc = RSCodec(ecc)

    # Expects serialized data
    def reed_solo_encode(self, data):
        return self.rsc.encode(data)

    # Returns serialized data
    def reed_solo_decode(self, data):
        decoded_data = self.rsc.decode(data)[0]
        # Logging if we corrected an error
        if not self.find_discrepancy(data, decoded_data):
            print("Message corrected")

        return decoded_data

    def find_discrepancy(self, byte_array, substring):
        return byte_array.find(substring) != -1
```

Listing 2: reed_solomon.py

A.3 Server Code

```
import logging
from socket import *
from _thread import start_new_thread
import time
import psutil
from helpers.constants import *
from network.reed_solomon import ReedSolomon
from objects import paddle, ball
import pickle

# No need for queue on server side since we
# will only receive and forward packets

# Global variables
HOST = 'localhost'
ADDR = (HOST, PORT)

```

```
corr = ReedSolomon(ECC)

server_socket = socket(AF_INET, SOCK_DGRAM)

inital_paddles = [paddle.Paddle(0, (WIN_HEIGHT - PADDLE_HEIGHT) / 2,
    PADDLE_WIDTH, PADDLE_HEIGHT, (255, 255, 255)), paddle.Paddle(WIN_WIDTH -
    PADDLE_WIDTH, (WIN_HEIGHT - PADDLE_HEIGHT) / 2, PADDLE_WIDTH,
    PADDLE_HEIGHT, (255, 255, 255))]
initial_ball = ball.Ball(WIN_WIDTH / 2, WIN_HEIGHT / 2, BALL_RADIUS, (255,
    255, 255))

paddles = {}    # Dictionary mapping paddles and addresses
balls = {}      # Dictionary mapping balls and addresses

def initialize(addr):

    if len(paddles) == 0:
        paddles[addr] = inital_paddles[0]
        balls[addr] = initial_ball
    elif len(paddles) == 1:
        currAddr = next(k for k, v in paddles.items() if k != addr)
        if paddles[currAddr].x == 0:
            paddles[addr] = inital_paddles[1]
            paddles[currAddr] = inital_paddles[0]
        else:
            paddles[addr] = inital_paddles[0]
            paddles[currAddr] = inital_paddles[1]
        balls[addr] = initial_ball
        balls[currAddr] = initial_ball
    else:
        # If we have more than 2 clients trying to connect
        return False
    return True

def client_thread(data, addr):

    if isinstance(data, paddle.Paddle):
        # We received a paddle
        paddles[addr] = data
        reply = next(v for k, v in paddles.items() if k != addr)
    elif isinstance(data, ball.Ball):
        # We received a ball
        balls[addr] = data
        reply = next(v for k, v in balls.items() if k != addr)
    else:
        reply = data

    encoded_data = corr.reed_solo_encode(pickle.dumps(reply))

    server_socket.sendto(encoded_data, addr)

    #print ("Received ", data)
    #print ("Sending ", reply)

def process_data(decoded_data, addr, log=False):
    if len(paddles) < 2 and not log:
        encoded_data = corr.reed_solo_encode(pickle.dumps("wait"))
```

```
        server_socket.sendto(encoded_data, addr)
        return False
    elif decoded_data == "init_paddle":
        encoded_data = corr.reed_solo_encode(pickle.dumps(paddles[addr]))
        server_socket.sendto(encoded_data, addr)
    elif decoded_data == "init_ball":
        encoded_data = corr.reed_solo_encode(pickle.dumps(balls[addr]))
        server_socket.sendto(encoded_data, addr)
    elif decoded_data == "end":
        paddles.pop(addr)
        balls.pop(addr)
    else:
        start_new_thread(client_thread, (decoded_data, addr))
    return True

def main():

    try:
        server_socket.bind(ADDR)
    except error as e:
        str(e)

    print("Waiting for data, server started")

    try:

        # Looping infinetely until we receive data
        while True:

            data, addr = server_socket.recvfrom(BUFSIZE)

            # Verifying if this address has sent data before
            if addr not in paddles:
                if not initialize(addr):
                    continue
                else:
                    print("Address", addr, "added")

            decoded_data = corr.reed_solo_decode(data)
            decoded_data = pickle.loads(decoded_data)

            if not process_data(decoded_data, addr):
                continue

    except KeyboardInterrupt:
        print("Server stopped")

def log():

    try:
        server_socket.bind(ADDR)
    except error as e:
        str(e)

    logging.basicConfig(filename='server.log', level=logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')

```

```
    try:
        start_time = time.time()  # Start time for calculating throughput
        total_data_received = 0

        while True:
            # Measure response time
            start_resp_time = time.time()

            data, addr = server_socket.recvfrom(BUFSIZE)
            decoded_data = corr.reed_solo_decode(data)
            decoded_data = pickle.loads(decoded_data)

            if not process_data(decoded_data, addr, True):
                continue

            # Logging response time
            end_resp_time = time.time()
            response_time = end_resp_time - start_resp_time
            logging.info(f'Response Time: {response_time}')

            # Accumulate length of data received
            total_data_received += len(data)

            # Measure resource utilization
            cpu_percent = psutil.cpu_percent()  # CPU utilization as a
    percentage
            mem_percent = psutil.virtual_memory().percent  # Memory
    utilization as a percentage
            logging.info(f'CPU Utilization: {cpu_percent}%')
            logging.info(f'Memory Utilization: {mem_percent}%')

    except KeyboardInterrupt:

        end_time = time.time()
        time_elapsed = end_time - start_time
        throughput = total_data_received / time_elapsed  # Total throughput
    in bytes per second
        logging.info(f'Throughput: {throughput} B/s')

        logging.info('Server stopped')


if __name__ == '__main__':
    #main()
    log()
```

Listing 3: server.py