P3 Middleware

Project report

A Distributed Systems Report

by Group 2:
Andreas Kaag Thomsen, 202105844
Buster Salomon Rasmussen, 202107215
Supervisors: Christian Fischer Pedersen & Christian Marius Lillelund

AARHUS UNIVERSITET

November 27, 2023

Contents

1 Introduction

The focus of this project is to design, implement and test a distributed version of the game 'pong' with real-time constraints to ensure smooth running of the game using message-oriented middleware (MOM).

MOM is a type of middleware that focused on exchanging data between recipients. MOM has many advantages, such as asynchronous and multi-point transmission, loosely coupling between participants, and high scalability and reliability. It simplifies data transfer and hides underlying heterogeneity in hardware systems. Today, there exist many different MOM implementations for various applications [11, p. 88].

In this project we chose to use RabbitMQ, which is an open source messaging system that implements the Advanced Message Queuing Protocol (AMQP) [11, p. 94]. The deciding benefits of using RabbitMQ was its diverse set of user API's including the python library `pika` and support for multiple messaging models including the pub/sub model.

2 Methods and tools

In this project the pong game has been implemented in object oriented Python. We have used Visual Studio Code as our integrated development environment (IDE) and Github for code sharing and version control. In the design phase we have used the unified modelling language (UML) to showcase the design and architecture of our implementation.

3 Results

In this section we present the game that has been developed. This is showcased mainly through diagrams expressing the flow of communication as well as the code structure and architecture. Moreover, we present selected code snippets that are most vital for the game.

3.1 Architecture

This section presents high-level architecture of the of game through diagrams. The architecture will be presented from three different views elaborated underneath.

- Communication view: Presents the **communication structure** in the system, that actors use to share information.
- Logical view: Presents the **static structure** of the code through class diagrams.
- Process view: Presents the **control-flow and interaction** between different objects in the system, through state- and sequence diagrams.

3.1.1 Communication view

For communication between the clients and the server, we have used RabbitMQ, which is a message-oriented middleware. In order to communicate, the application first sets up a connection. A connection is a TCP-based communication-channel on the application layer. Specifcally, RabbitMQ implements the Advanced Message Queueing Protocol (AMQP). The connection represents a TCP-connection between a *client* and the RabbitMQ broker. After the connection has been authenticated by the server, the client can configure its *channels*, which can be viewed as a logical connection on the physical TCP-connection. Inside a connection there can be configured several RabbitMQ channels allowing for connection multiplexing [1] [8]. Usually, each thread has its own channel, and there exist only one connection per process [7]. When the above configuration is done, AMQP uses the *AMQP entities* queues, bindings and exchanges in its model in the following way: Messages are published to exchanges, which then routes messages to a queue based on the bindings [6]. Our specific communication design is depicted in Figure 1.

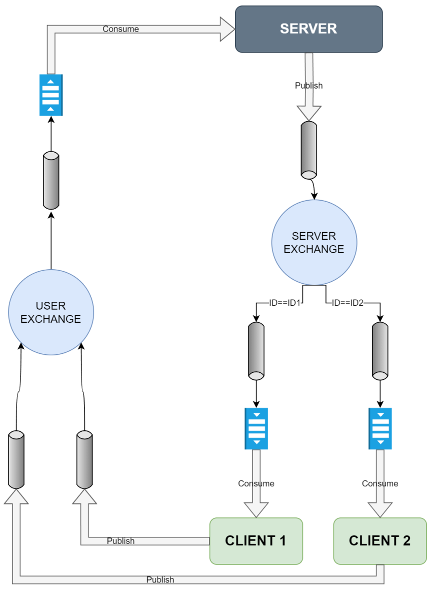


Figure 1: The communication design

Messages from the server to the clients are published to the `SERVER_EXCHANGE` and routed to a player's queue according to each player's unique `player_id`. Vice versa, messages from any player are published to the `USER_EXCHANGE` and routed directly to the server's queue without any routing key.

3.1.2 Logical view

On the player side, we have utilized the Model-View-Controller (MVC) design pattern. In this design pattern, the model represents knowledge and encapsulates the application state. This includes paddle positions, ball positions and scores. The view is a visual representation of the model and acts as a presentation filter, by highlighting certain aspects of the model and suppressing others. Our view uses `turtle` as graphics library. The controller deals with user inputs and modifies the model. It receives such inputs and acts accordingly by presenting relevant views [10].

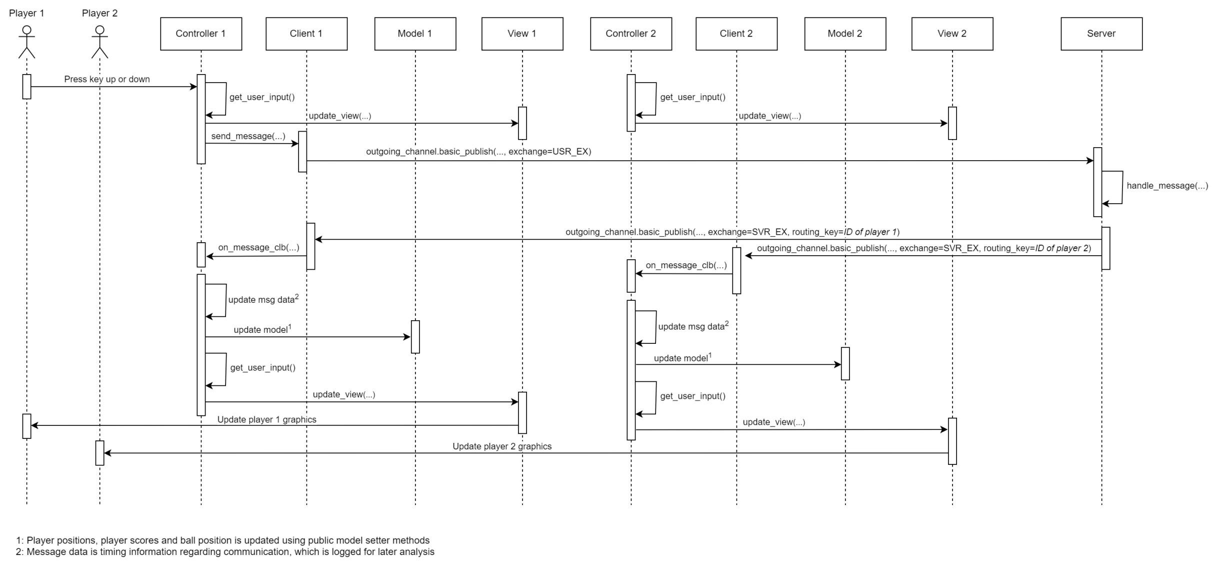
On the server side, there is only one class called Server, which is responsible for everything. That includes getting client updates, updating its own internal model, and sending out new game updated game states.

Due to the large size of the class diagrams, they are placed in Appendix A.

[1]It is recommended not to share the physical TCP-connection between consuming and publishing channel [8] such that the consumer is not affected by a busy publisher.

3.1.3 Process view

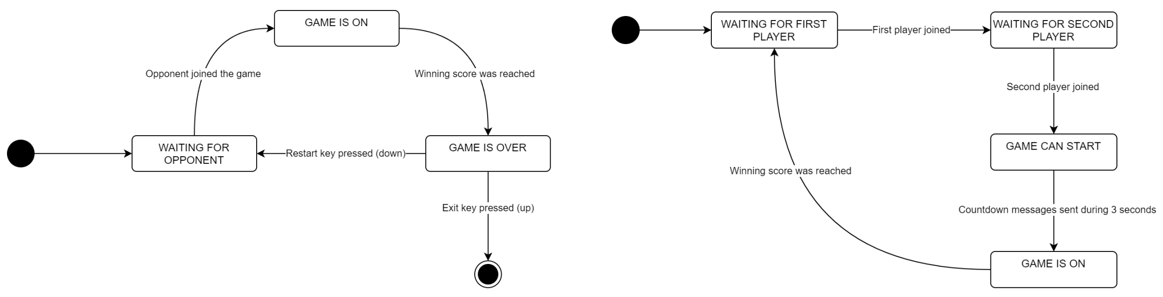
Figure 2 shows a simplified sequence diagram. This scenario highlights the majority of interactions between entities of the system when one player tries to move a paddle by pressing their corresponding up/down key.



1: Player positions, player scores and ball position is updated using public model setter methods
2: Message data is timing information regarding communication, which is logged for later analysis

Figure 2: Simplified sequence diagram encapsulating interaction between system entities upon paddle movement request by a user. Horizontally shifted activation bars, that are associated with the same object, represents unique threads.

Figure 3a and Figure 3b depicts state diagrams for the controller and server respectively. Both state diagrams presents implicit state interactions of the subsystems in the sense that this is the high-level logic that are being executed. It is however not implemented explicitly as there are no such state variables in the code.



(a) States in `controller.py`

(b) States in `server.py`

Figure 3: State diagrams

3.2 Design & Implementation

This section provides an insight into the source code that presents how we have implemented the logic presented in the preceding section.

3.2.1 Use of middleware

One of the most vital aspects of this game is the communication between the clients and the server. For this reason, we show how the design in Figure 1 is implemented using the Python module `pika`. The code snippets shown below are taken from `server.py` since the configuration on the client side is quite similar.
To create a channel, we execute the following code:

```
connection = pika.BlockingConnection(
pika.ConnectionParameters(host=RMQ_CONFIG.SERVER_IP, port=RMQ_CONFIG.SERVER_PORT))
channel = connection.channel()
return channel
```

For each client and the server, two channels are created - one incoming and one outgoing. Underneath, the outgoing channel is configured. It is declared what exchange the channel should use and what the exchange type should be. Since the messages to each player should be delivered only to one of them, the exchange type is `direct` for the server. In turn, each client binds its queue to the routing key that is to be used by the server. This routing key is each player's unique player id.

```
self.outgoing_channel.exchange_declare(
            exchange=RMQ_CONFIG.SERVER_EXCHANGE, exchange_type='direct')
```

The incoming channel is configured in the following way:

```
"""Configure the consumer channel"""
self.incoming_channel.exchange_declare(
    exchange=RMQ_CONFIG.USER_EXCHANGE, exchange_type='fanout')  # for incomming messages
# Incoming message queue
result = self.incoming_channel.queue_declare(queue='', exclusive=True)
self.incoming_message_queue = result.method.queue
# Bind the queue to the exchange
self.incoming_channel.queue_bind(
    exchange=RMQ_CONFIG.USER_EXCHANGE, queue=self.incoming_message_queue)
# Create a consumer for the incoming message queue
self.incoming_channel.basic_consume(
    queue=self.incoming_message_queue, on_message_callback=self.on_message, auto_ack=True)
```

Again, we first declare what exchange should be used. Next, we declare the queue that the server consumes on and binds this to the proper exchange. Lastly, we setup the consumer where we provide the newly created queue as well as a callback method, which is invoked each time a new message arrives in the queue. In order to start consuming we call the method `start_consuming` on the incoming channel. This is a blocking function and thus it runs it its own thread continually checking for incoming messages, as it was also depicted in the sequence diagram in Figure 2.

3.2.2 Game logic

On the server side, the main logic is executed by the method `state_thread_fun`. This method runs in a dedicated thread, and periodically sends out player updates. Since the body of this method is quite long, we only show it in pseudo code in Listing 1.

```python
def state_thread_fun():
    while not state_thread_stop_event.is_set():
        while game_is_on:
            calculate_ball_pos()
            ball_state, payload = collision.determine_game_state(...)
            if(ball_state == BALL_STATE.COLLISION):
                d_ball = payload
            increment_goals(ball_state)
            send_player_updates()
            time.sleep(1/refresh_rate)
        if (game_is_done):
            reset_model()
```

Listing 1: The method `state_thread_fun` from `server.py` shown in pseudo code

As long as the stopping event has not been set, the thread runs periodically. When the variable `game_is_on` is set to true (i.e. when two players have joined the game), the main body of the thread is executed. For each iteration, it starts by calculating the newest ball position. Next, it determines the new game state by providing the newest game parameters to the method `determine_game_state()`. Afterwards, it acts according to the new game state and sends out player updates with information about player and position as well as the game scores.

On the player side, the method `main_game_loop` handles the majority of the game logic. That is, the logic being executed in the state `GAME IS ON` as shown in Figure 3a. To avoid unnecessary complexity, this is shown in a simplified version in Listing 2.

```
1  while not self.stop_main_loop.is_set():
2      while self.game_is_on:
3          if not self.incoming_message_queue.empty():
4              msg = self.incoming_message_queue.get()
5              self.handle_message(msg)
6          self.game_view.update_view(self.get_model())
7          if self.game_finished:
8              self.game_view.show_winner(self.winner)
9              self.game_is_on = False
10             break
11         dt = self.get_user_input()
12         if dt != 0:
13             self.client.send_message(
14                 (message_parsing.encode_message(MSG_TYPES.PLAYER_UPDATE_USR,
15                                                 self.player_id,
16                                                 self.calculate_msg_id(),
17                                                 dt)))
18     self.stop_game()
```

Listing 2: The body of the method `main_game_loop()` from `controller.py` (simplified version)

For each iteration of the game-loop, we start by checking for any server updates. If there is any, this message is handled according to the message type by the method `handle_message`. Next, the view is updated based on the newest model parameters and we check if the game is finished. This variable is updated within `handle_message`. If the game has finished we break out of the main game loop and jump to line 18, which makes the state transition to `GAME IS OVER` as shown in Figure 3a. Finally, any user input is taken and sent to the server via the client.

4 Experiments

In this section we elaborate on the experiments we have conducted to test the game.

4.1 Experiment setup

In order to test the game, we have started two instances of `player_main.py` in two different terminals and one instance of `server_main.py` in a third terminal. These scripts simply create a `Controller(key_up, key_down)` object and a `Server()` object, respectively. Doing this creates the setup as shown in Figure 4.

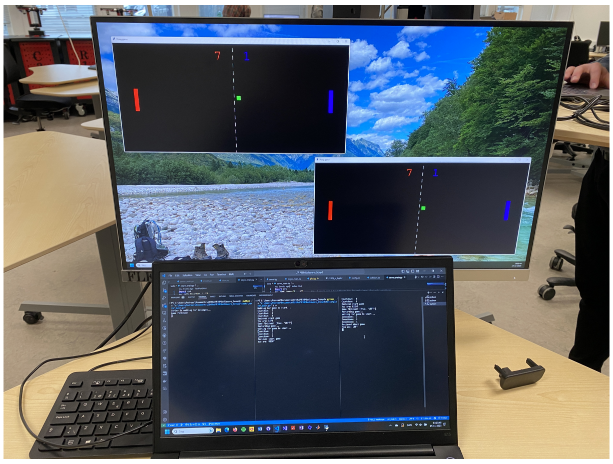


Figure 4: Experiment setup

With this setup, we were able to qualitatively test the pong game by simply playing it. All game functionalities work as expected but the game is quite slow as elaborated next.

4.2 Message transmission time

In order to quantify the quality of the game, we have chosen to measure the *message transmission time*. We define this as the time it takes from a player sends a player update to the server and until the player receives a message from the server with updated game info. This corresponds to the response time of the system. We have done this by appending a unique message id to each message sent from the client and saving this id with a timestamp. When the server handles that message it re-transmits the message id such that the client can calculate the time passed. We have made the plot shown in Figure 5. These plots are based on all measurements from 4 finished games.

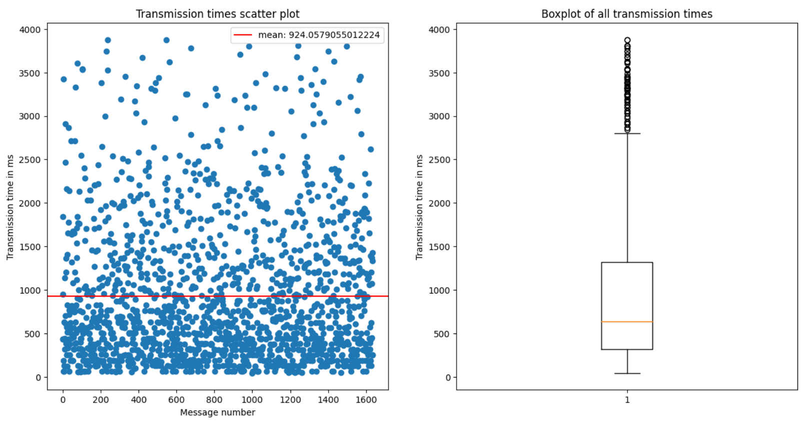


Figure 5: Message transmission times based on 4 games

We observe that the data is very dense in the low range. The boxplot to the right also shows that the interquartile range is between $\sim$ 300 ms and $\sim$ 1300 ms and a quite long whisker towards higher values. In addition, we also see some severe outliers. We conclude that in average the system performs well with response times around 0.5 s but in some cases the response time is unacceptedly high for reasons we cannot explain. However, holding the quantitative and qualitative analysis together, we find that the game performs sufficiently since it gives an adequate gaming experience.

5 Discussion

We were able to make a working distributed pong game. However, we have noticed that optimizations could be made when it comes to constraints specification, message transmission time, project complexity and structure. These will be discussed underneath.

5.1 Constraints specification

We have not made any real-time time constraints prior to the development of the game though the problem description stated we should. This led to an arbitrary choice of messaging middleware, which resulted in a slow game, as discussed next. Our reasoning behind not setting any constraint was simply lack of experience so we did not have any expectation of what would be reasonable.

5.2 Optimizing message transmission time

The game is evidently not very fast, as documented in the experiment section. Comparing with similar implementations such as the one in [4], our game is quite slow. We suspect that this is mainly due to the use of RabbitMQ, which is a TCP-based protocol [8]. The article in [2] investigates the use of TCP in MMORPGs[2] and finds that *TCP is unwieldly and inappropriate for MMORPGs.* They find that since the game packets are generally small, the TCP/IP header takes up to 46% of the total bandwidth used. Moreover, TCP's congestion control mechanism proves to be ineffective and leads to additional latency when a burst of commands is generated. This is also the case in our game whenever a player holds down a key. Furthermore, they conclude that TCP is overkill as not every game packet needs to be transmitted reliably and ordered. Though the research in [2] is based on MMORPGs, the game characteristics making the TCP connection ineffective are essentially shared by all real-time interactive network games [2] including Pong. Based on the findings above, we have not chosen the optimal networking protocol for our implementation. Better choices of transport layer protocols would be the the *User Datagram Protocol (UDP)* or *Reliable User Datagram Protocol (RUDP).* Today, these are the primary choices for network game companies, and especially RUDP has gained traction since 2015 [3]. UDP offers high speeds but it is not reliable [9]. RUDP on the other hand offers higher reliability than UDP, by extending it with acknowledgement, flow control, retransmission and over-buffering, and also promises faster speeds than TCP, by trading off congestion control [1]. Thus, RUDP can be seen as a good compromise between UDP and TCP. Examples of UDP-based middleware libraries in python we could use are `socket` or `twisted`. There also exist some python implementation of RUDP on the internet, but the quality is not guaranteed to be great.

Also, we want to highlight the severe outliers in Figure 5 that we cannot necessarily blame on the use of TCP. We are still not able to explain why these occur, but it may be caused by the way we conduct the

[2]Massive Multiplayer Online Role Playing Games

measurements.

Additionally, we are not convinced that we have used the message broker in the most efficient way possible. More specifically, we might not have chosen the right connection adapter for the purpose. We chose to use the `Blocking Connection Adapter`, but according to pika's documentation on this, other adapters, such as the `Select Connection Adapter`, are faster in some ways and better suited for handling asynchronous events [5].

5.3 Project complexity

Moreover, we have noticed two areas of improvements with regards to project complexity and structure. The first being that the states are not explicitly implemented in the code, which makes the code more complex to understand. Arguably, an explicitly implemented state-machine would help manage complexity. The second being to separate logic and data on the server side, instead of both being managed inside the server class. This class could be split in two using an MC (Model and Controller) pattern.

6 Conclusion

In this project, we have designed, implemented and tested a distributed version of the pong game using RabbitMQ as the message-oriented middleware. We have used Python as the programming language and pika as the RabbitMQ library. We have followed a communication model based on exchanges and queues, and a logical model based on the MVC pattern. We have also used UML diagrams to illustrate the architecture and the interactions of our system. We have conducted experiments to measure the message transmission time and the quality of the game. We have found that our game performs well in average, but suffers from some latency issues and outliers. We have discussed the possible causes and solutions for these problems, and suggested some improvements for the future work. We have learned how to use RabbitMQ and pika to create a distributed system, and how to apply the concepts of middleware, messaging, and concurrency to a real-time interactive network game.

7 References

[1] Tom Bova and Ted Krivoruchka. *RELIABLE UDP PROTOCOL*. Internet-Draft draft-ietf-sigtran-reliable-udp-00. Work in Progress. Internet Engineering Task Force, Feb. 1999. 15 pp. URL: https://datatracker.ietf.org/doc/draft-ietf-sigtran-reliable-udp/00/.

[2] Kuan-Ta Chen et al. "An Empirical Evaluation of TCP Performance in Online Games". In: ACE '06. Hollywood, California, USA: Association for Computing Machinery, 2006, 5–es. ISBN: 1595933808. DOI: 10.1145/1178823.1178830.

[3] Jun-Ho Huh. "Reliable User Datagram Protocol as a Solution to Latencies in Network Games". In: *Electronics* 7.11 (Nov. 2018), p. 295. ISSN: 2079-9292. DOI: 10.3390/electronics7110295. URL: http://dx.doi.org/10.3390/electronics7110295.

[4] Perforated Software. *pong-2*. URL: https://pong-2.com/. (accessed 23.11.2023).

[5] pika. *Connection Adapters*. URL: https://pika.readthedocs.io/en/stable/modules/adapters/index.html. (accessed 15.11.2023).

[6] RabbitMQ. *AMQP Concepts*. URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html. (accessed 27.11.2023).

[7] RabbitMQ. *Connection Adapters*. URL: https://www.rabbitmq.com/channels.html. (accessed 24.11.2023).

[8] RabbitMQ. *Connections*. URL: https://www.rabbitmq.com/connections.html. (accessed 24.11.2023).

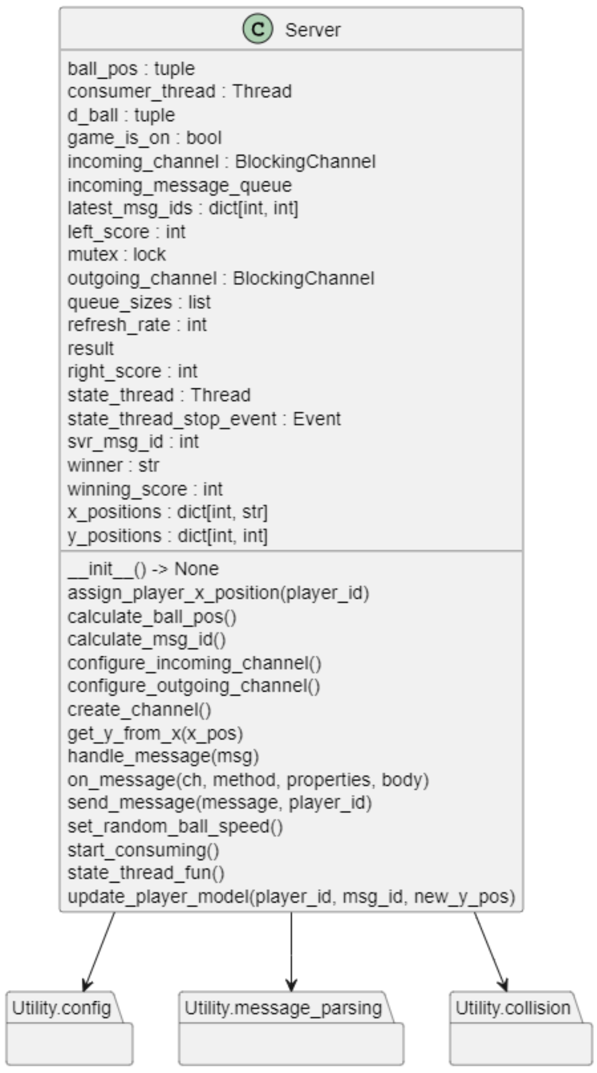
[9] *User Datagram Protocol*. RFC 768. Aug. 1980. DOI: 10.17487/RFC0768. URL: https://www.rfc-editor.org/info/rfc768.

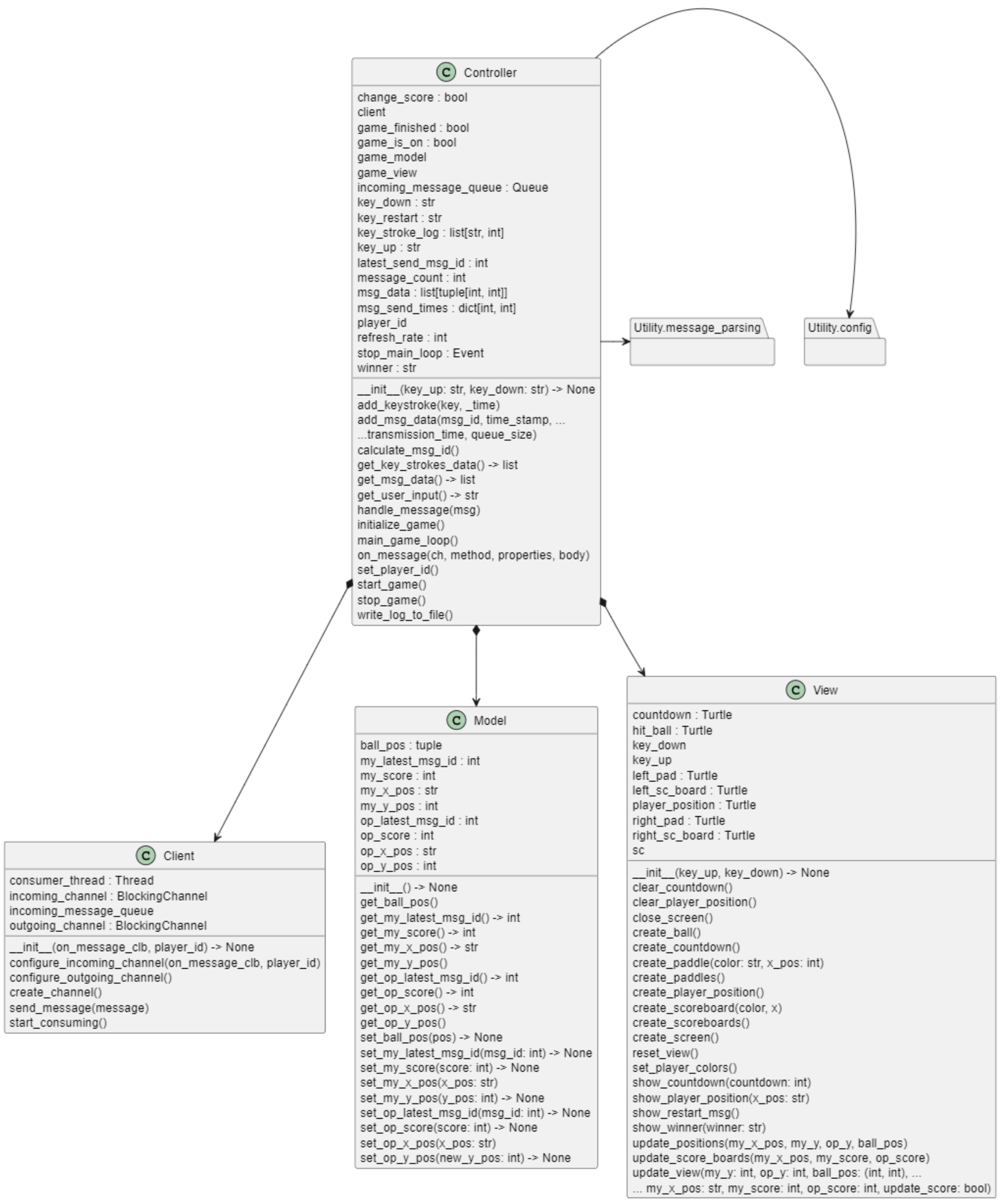
[10] Stephen Walther. *The evolution of MVC*. URL: http://stephenwalther.com/archive/2008/08/24/the-evolution-of-mvc. (accessed: 16.5.2023).

[11] Jiang Yongguo et al. “Message-oriented Middleware: A Review”. In: *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*. 2019, pp. 88–97. DOI: 10.1109/BIGCOM.2019.00023.

8 Appendices

8.1 Appendix A




C Controller
change_score : bool
client
game_finished : bool
game_is_on : bool
game_model
game_view
incoming_message_queue : Queue
key_down : str
key_restart : str
key_stroke_log : list[str, int]
key_up : str
latest_send_msg_id : int
message_count : int
msg_data : list[tuple[int, int]]
msg_send_times : dict[int, int]
player_id
refresh_rate : int
stop_main_loop : Event
winner : str
__init__(key_up: str, key_down: str) -> None
add_keystroke(key, _time)
add_msg_data(msg_id, time_stamp, ...
...transmission_time, queue_size)
calculate_msg_id()
get_key_strokes_data() -> list
get_msg_data() -> list
get_user_input() -> str
handle_message(msg)
initialize_game()
main_game_loop()
on_message(ch, method, properties, body)
set_player_id()
start_game()
stop_game()
write_log_to_file()
Utility.message_parsing
Utility.config
C Client
consumer_thread : Thread
incoming_channel : BlockingChannel
incoming_message_queue
outgoing_channel : BlockingChannel
__init__(on_message_clb, player_id) -> None
configure_incoming_channel(on_message_clb, player_id)
configure_outgoing_channel()
create_channel()
send_message(message)
start_consuming()
C Model
ball_pos : tuple
my_latest_msg_id : int
my_score : int
my_x_pos : str
my_y_pos : int
op_latest_msg_id : int
op_score : int
op_x_pos : str
op_y_pos : int
__init__() -> None
get_ball_pos()
get_my_latest_msg_id() -> int
get_my_score() -> int
get_my_x_pos() -> str
get_my_y_pos()
get_op_latest_msg_id() -> int
get_op_score() -> int
get_op_x_pos() -> str
get_op_y_pos()
set_ball_pos(pos) -> None
set_my_latest_msg_id(msg_id: int) -> None
set_my_score(score: int) -> None
set_my_x_pos(x_pos: str)
set_my_y_pos(y_pos: int) -> None
set_op_latest_msg_id(msg_id: int) -> None
set_op_score(score: int) -> None
set_op_x_pos(x_pos: str)
set_op_y_pos(new_y_pos: int) -> None
C View
countdown : Turtle
hit_ball : Turtle
key_down
key_up
left_pad : Turtle
left_sc_board : Turtle
player_position : Turtle
right_pad : Turtle
right_sc_board : Turtle
sc
__init__(key_up, key_down) -> None
clear_countdown()
clear_player_position()
close_screen()
create_ball()
create_countdown()
create_paddle(color: str, x_pos: int)
create_paddles()
create_player_position()
create_scoreboard(color, x)
create_scoreboards()
create_screen()
reset_view()
set_player_colors()
show_countdown(countdown: int)
show_player_position(x_pos: str)
show_restart_msg()
show_winner(winner: str)
update_positions(my_x_pos, my_y, op_y, ball_pos)
update_score_boards(my_x_pos, my_score, op_score)
update_view(my_y: int, op_y: int, ball_pos: (int, int), ...
... my_x_pos: str, my_score: int, op_score: int, update_score: bool)