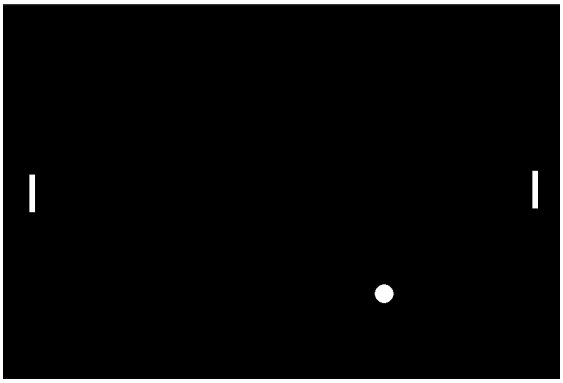
Distributed Pong: Adding Communication Protocols and Etcd State Management

Lorenzo Massone
lorenzo.massone@studio.unibo.it

November 2024



Distributed Pong is an educational project that implements the classic Pong game in a distributed environment, showcasing various networking patterns and distributed system concepts. The project adopts a flexible architecture centered around a coordinator-terminal pattern, where a coordinator (server) manages the overall game state and synchronizes multiple terminals (clients). This report documents additional implementations of communication protocols such as ZeroMQ and WebSockets (with a RESTful API for game start coordination), and an alternative state management solution using etcd with a basic leader election mechanism and state synchronization using events pushed on the etcd database.

1 Goal/Requirements

The goals for this project were to integrate additional communication protocols into the existing project, which previously only supported UDP for communication. The required integrations were:

- Implementing ZeroMQ as an alternative communication protocol for communication between the coordinator and terminals.
- Implementing WebSockets as an alternative communication protocol for communication between the coordinator and terminals. The connection to the WebSocket server should be managed with a RESTful API to make the game start only when all players are connected.
- Implementing an alternative solution with etcd to manage the state of the game without a coordinator while keeping the state consistent

1.1 Scenarios

The possible scenarios for the project are:

- Playing Pong using a centralized coordinator that manages the game state and synchronizes the players, while offering choice between the various communication protocols

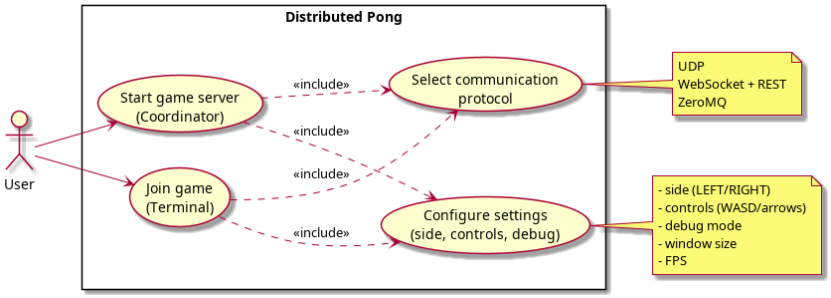


Figure 1: Centralised version of the game

- Playing Pong using etcd to manage the game state without a coordinator.

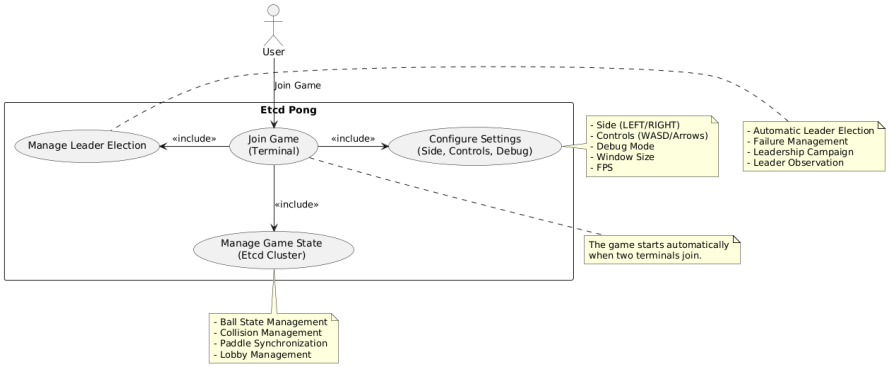


Figure 2: Etcd implementation of the game

1.2 Self-assessment policy

- Reliability of ZeroMQ message passing as alternative communication protocol.
- WebSocket lobby management with the REST API for coordinated game starts.
- Leader election and state consistency in the etcd implementation.
- Protocol switching between UDP, ZeroMQ, and WebSockets to verify interface compliance.

The emphasis was placed on making each protocol implementation clear and well-structured to serve as educational examples of different distributed system approaches. Thus, the quality of the code was assessed based on its clarity, modularity, and adherence to the already established project structure.

2 Requirements Analysis

The implementation of these new communication protocols for educational purposes required maintaining the existing project structure and adding the new features in a way that would not disrupt the existing codebase. It was also fundamental to keep the code as simple and understandable as possible.

We chose to implement these technologies to demonstrate how different protocols can have varying advantages and disadvantages depending on the key attributes sought in a distributed system. The tradeoffs associated with each technology will be addressed in detail in a later section of this report.

3 Design

The system implements two distinct architectures for distributed gameplay: a centralised coordinator-terminal architecture and a decentralised etcd-based architecture. In the

coordinator-terminal architecture, components communicate through well-defined interfaces using different communication protocols (UDP, ZMQ, or WebSockets). The game state is managed by a `Coordinator` updating state according to the events produced by `Terminals`. In the etcd-based architecture, `Terminals` coordinate through a distributed store instead of using a central coordinator and a basic leader election system.

The architecture of the project is described in the following diagram:

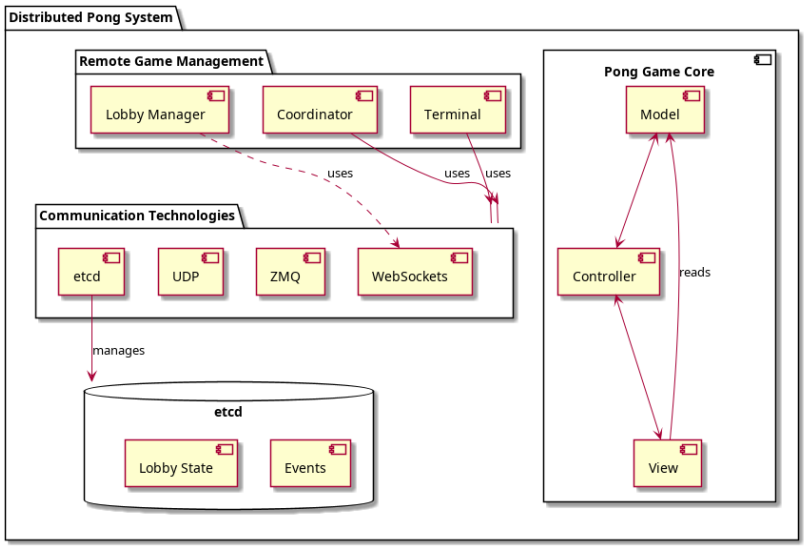


Figure 3: Basic view of the main logical components of the system

3.1 Structure

3.1.1 Remote Game Management

LobbyManager is responsible for handling player connections and game session management. It provides a REST API for players to create and join game sessions, maintaining the list of active players and their roles in each game. Importantly, the LobbyManager is exclusively used with WebSocket communication protocol, and is not available when using UDP, ZMQ, or etcd-based implementations.

Coordinator acts as the central game server in the coordinator-terminal pattern (UDP, ZMQ, WebSockets), orchestrating the game state and synchronizing communication between connected terminals. It manages the game loop and ensures consistent game state across all clients. This component is not present in the etcd-based implementation, where state management is handled through etcd directly.

Terminal represents a client node where players can interact with the game. Each terminal handles player input, renders the game state, and either communicates with the coordinator (in coordinator-terminal pattern) or directly with etcd (in

etcd-based pattern). In WebSocket mode, terminals must first interact with the LobbyManager to join a game session.

3.1.2 Communication Technologies Implemented

etcd provides an alternative implementation pattern for distributed gameplay, acting as a distributed key-value store. Instead of using a coordinator, terminals communicate by sharing state through etcd, as implemented in `dpongpy/etcd/etcd_pong_terminal.py`. This represents a different architectural approach from the coordinator-terminal pattern.

UDP implementation provides fast, lightweight communication between game components in the coordinator-terminal pattern. While not guaranteeing delivery, it's suitable for real-time game state updates where occasional packet loss is acceptable. UDP mode uses direct connections without a lobby system.

ZMQ offers a message queuing system that provides reliable communication patterns within the coordinator-terminal pattern. It supports different communication models (pub/sub, request/reply) and ensures message delivery. Like UDP, ZMQ mode doesn't utilize the lobby system and uses direct connections.

WebSockets combined with REST APIs enables bi-directional communication between clients and server in the coordinator-terminal pattern. WebSockets maintain persistent connections for real-time updates, while REST handles game session management. This is the only protocol that integrates with the lobby system.

3.1.3 Pong Game Core

These components remain consistent across both implementation patterns:

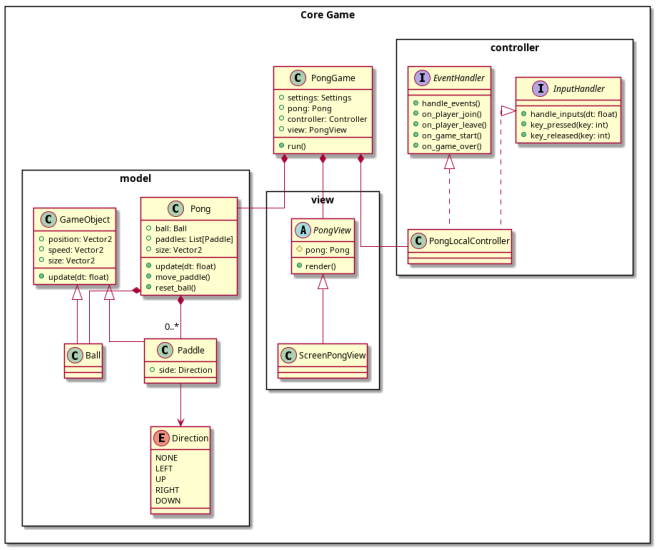


Figure 4: Basic view of the main logical components of the system

Model contains the core game logic and state management. It handles game physics, collision detection and maintains the game's rules and boundaries.

Controller processes player inputs and game events. It translates player actions into game state changes and communicates these changes either through the selected network protocol (coordinator-terminal pattern) or through etcd.

View handles the game's visual representation. It renders the game state (paddles, ball) and provides visual feedback for player actions.

3.1.4 Etcd

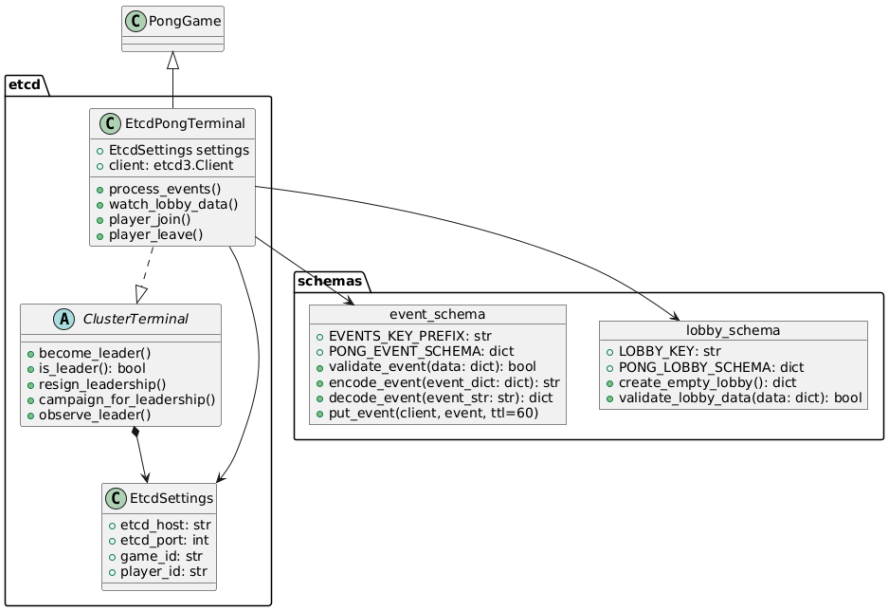


Figure 5: The `etcd` package

EtcdPongTerminal Represents the clients that connect to play the game using `etcd` as the coordination mechanism.

ClusterTerminal Represents all the logic related to lobby management, leader election, and how the leader processes events (modeled in `event_schema`) and updates the state of the game represented in `lobby_schema`.

lobby_schema A comprehensive JSON schema that unifies both the game and the session state. The schema includes:

- Game session information (gameId, gameState)
- Connected players array with their positions and directions
- Ball state including position and velocity

This unified schema is stored under a single key (`pong_lobby`) in `etcd`, acting as the source of truth for the state of the game.

event_schema Defined in `event_schema.py` as a JSON schema specifying the structure for transient game events such as player movements, collisions, and game state changes. These events are used for communication between terminals rather than persistent storage.

This schema-based approach provides flexibility in how the data is actually stored and processed, while ensuring data consistency through schema validation. The implemen-

tation in `etcd_pong_terminal.py` uses these schemas to validate and process the data stored in etcd.

3.1.5 Remote

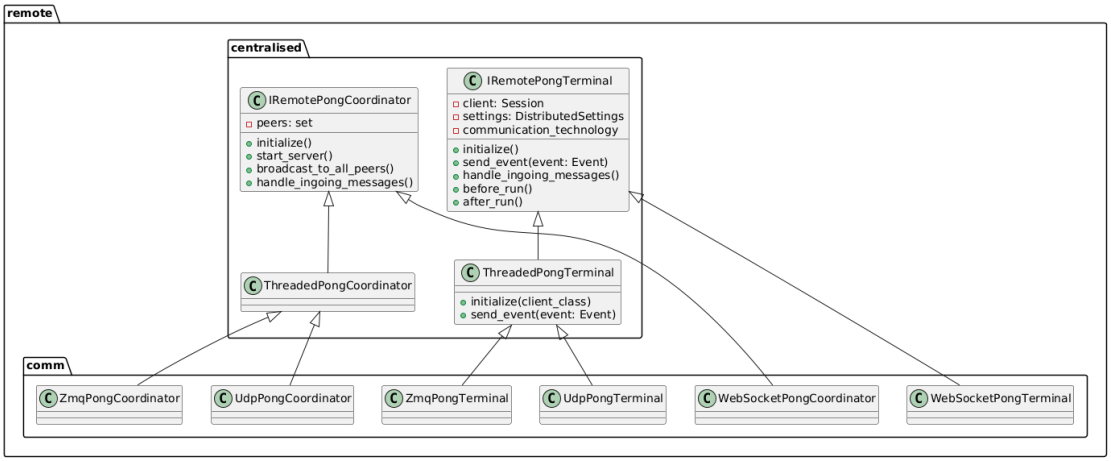


Figure 6: The `centralised` package

The main classes of the `centralised` package are:

IRemotePongCoordinator models the generic operations that a Coordinator should be able to do without considering any constraints linked to a specific communication protocol

IRemotePongTerminal represents the operations that a Terminal in a centralissed version should be able to do without considering any constraint linked to a specific communication protocol

ThreadedPongCoordinator generalizes the behaviour of the ZMQ and UPD implementations of the Coordinator since they have several similarities in the implementation

ThreadedPongTerminal generalizes the behaviour of the ZMQ and UPD implementation of the Terminal since they have several similarities in the implementation

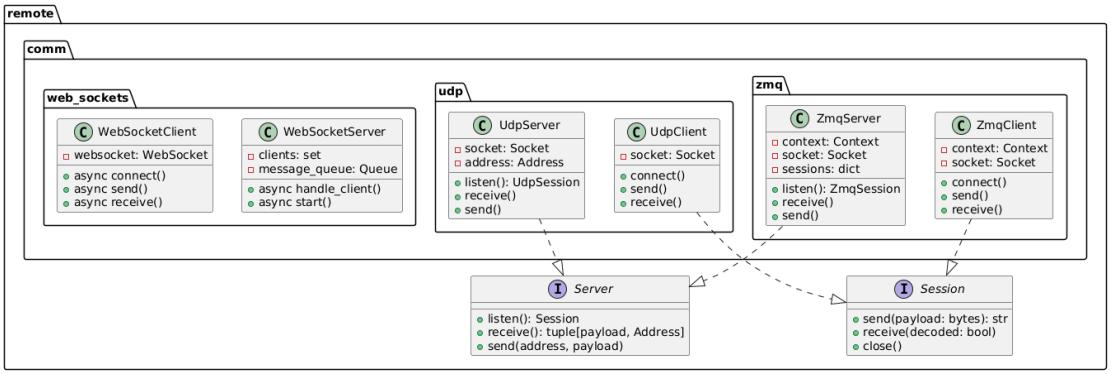


Figure 7: The `comm` package

The main classes of the `comm` package are:

ZmqPongCoordinator implements the Coordinator using the ZMQ communication protocol

ZmqPongTerminal implements the Terminal using the ZMQ communication protocol

UdpPongCoordinator implements the Coordinator using the UDP communication protocol

UdpPongTerminal implements the Terminal using the UDP communication protocol

WebSocketPongCoordinator implements the Coordinator using WebSockets as the communication protocol

WebSocketPongTerminal implements the Terminal using WebSockets as the communication protocol

Server is an interface representing the API followed by the coordinators

Session is an interface representing the API followed by the terminals

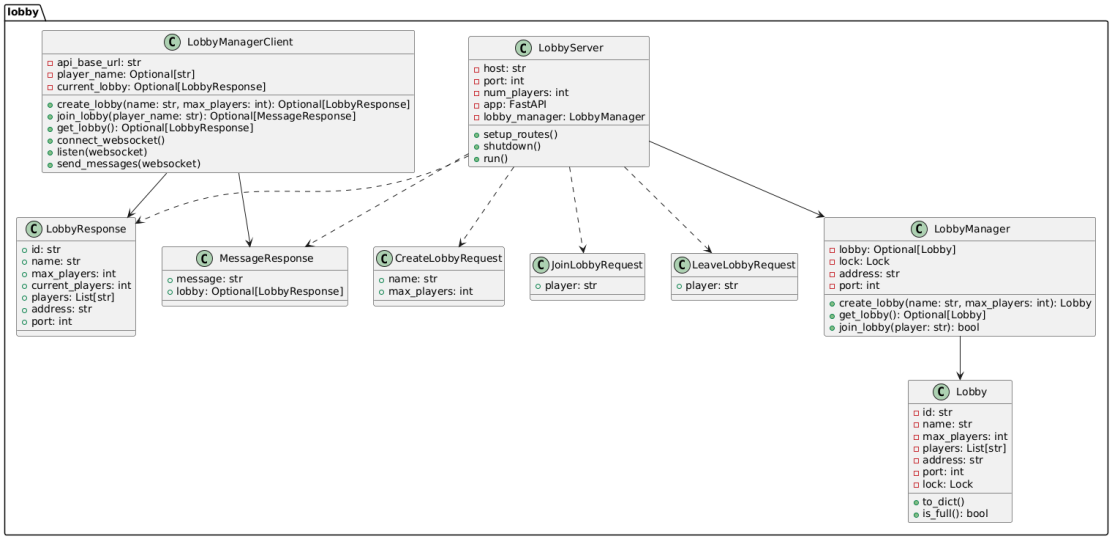


Figure 8: The `lobby` package

The main classes of the `lobby` package are:

LobbyManagerClient implements the code needed to interact with the RESTful API of the coordinator

LobbyServer implements the RESTful API needed for the lobby creation and management

LobbyManager implements core business logic for lobby lifecycle, like single lobby management and thread-safe async operations

Lobby implements a representation of the a game lobby

3.2 Behaviour

3.2.1 Centralised architecture

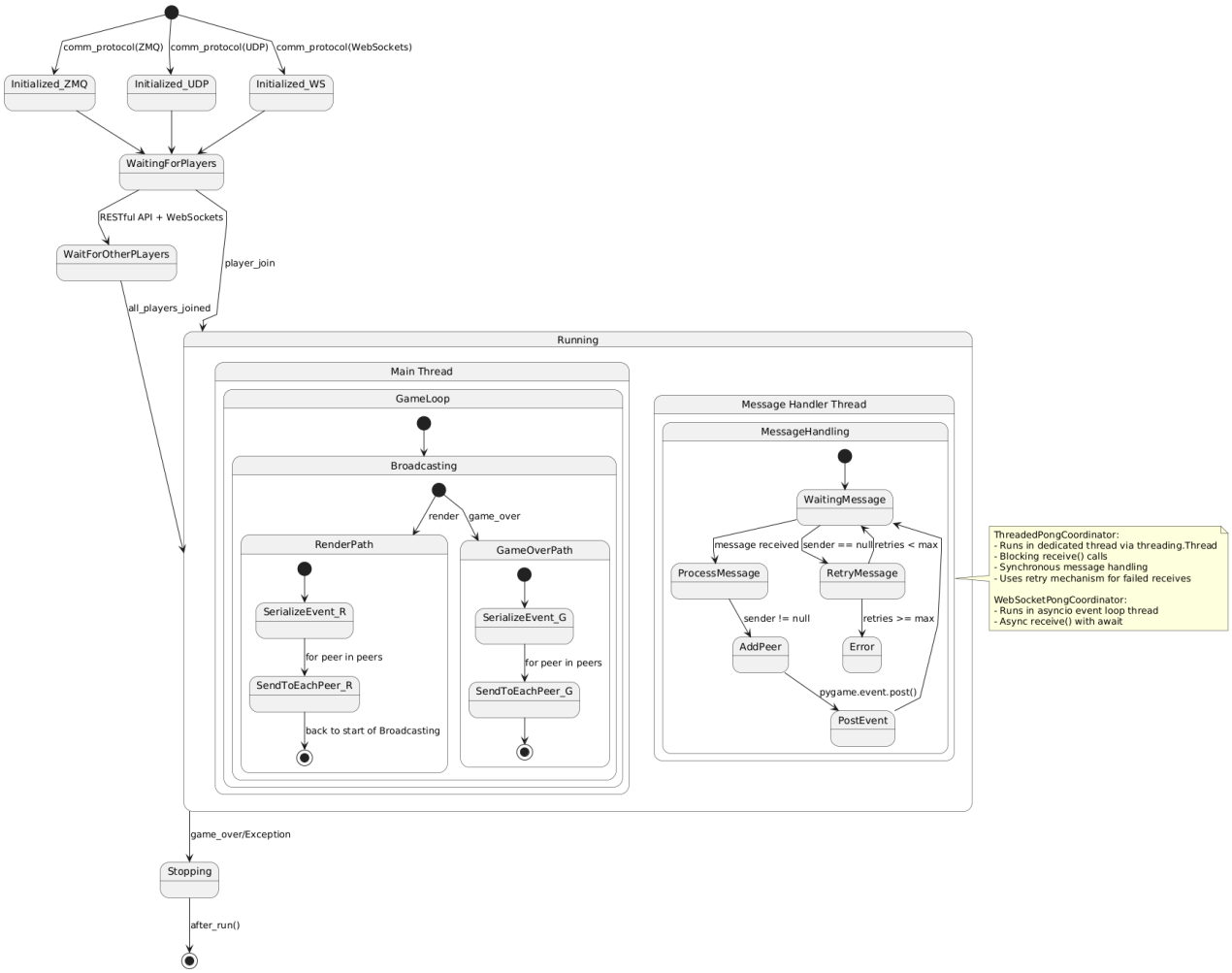


Figure 9: State diagram representing the behaviour of the `Coordinator` in the `centralised` architecture

When starting the `Coordinator` , the behavior changes based on the communication protocol chosen. In the case where the chosen communication protocol is `WebSockets` , the Coordinator waits for the set number of players to join the game using the RESTful API for lobby management. Otherwise, if the communication protocol is `ZMQ` or `UDP` the game starts after one player joins but the `GameLoop` resets after another player joins. After reaching the `Running` state, the control flow splits into two different threads:

- **GameLoop**: the state managed by the MainThread which manages the render of the view and the consequent broadcast of the elapsed time to the terminals to

update their local representations;

- **MessageHandlerThread**: this thread handles the incoming messages and processes them to post the corresponding events so that the local state is updated accordingly

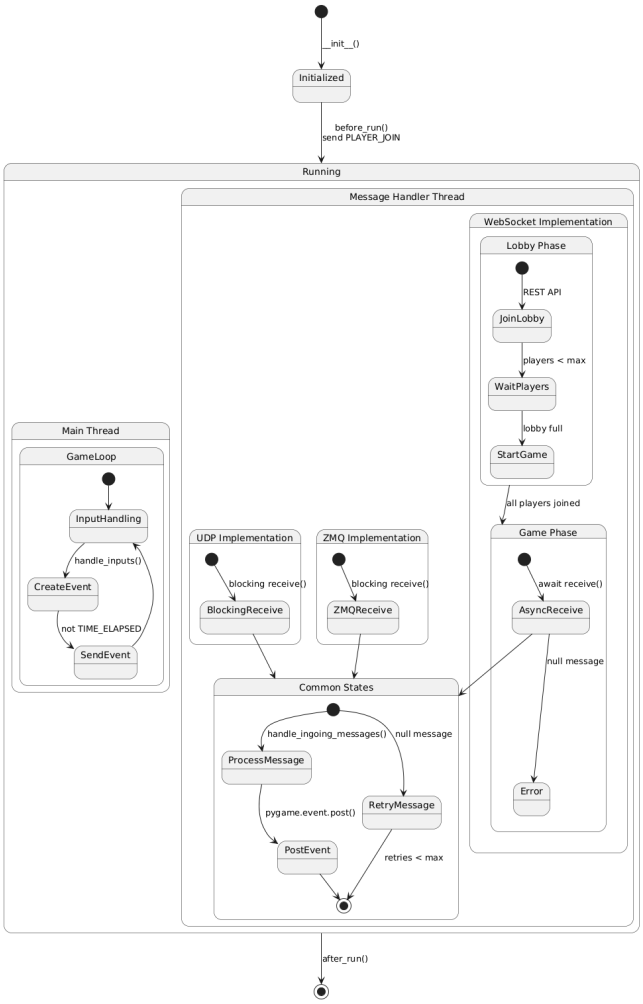


Figure 10: State diagram representing the behavior of the `Terminal` in the `centralized` architecture

From the client side, the control flow is quite similar to the one in the `Coordinator`. In the **MessageHandlerThread**, the `Terminal` manages the handling of the messages based on the communication protocol chosen at runtime. The messages then are handled to post the correct event so that the local state will be updated accordingly, while in the **MainThread** the input events are properly handled.

3.2.2 Etcd-based architecture

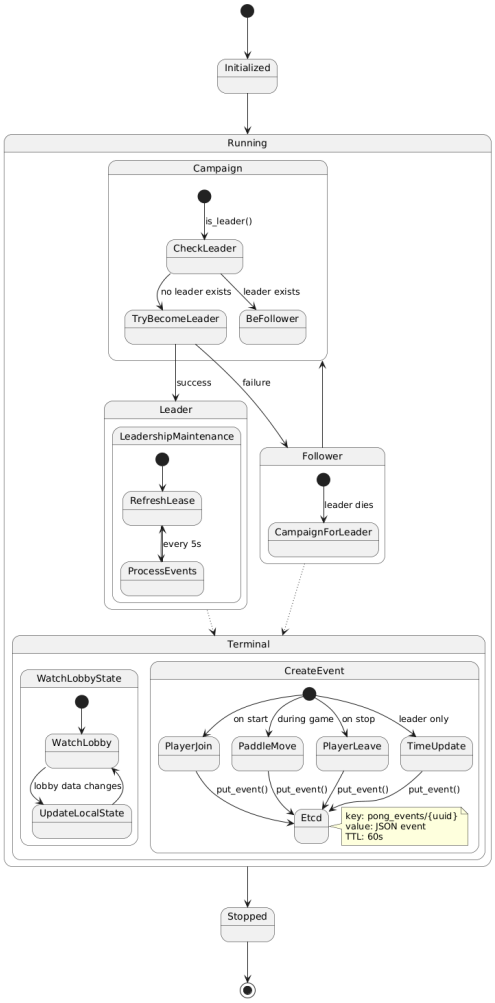


Figure 11: State diagram representing a `Terminal` in the etcd-based architecture

In the etcd-based architecture, the state is managed using the etcd database. The data representing the current state of the game are stored in the etcd database under the key `pong_lobby`. The format of the data is defined using the library `jsonschema` to enforce structure. Additionally, the events generated by both terminals are stored in the etcd database under the key `pong_events`. Since there is no `Coordinator` in this architecture, the current state of the program is managed by a **leader**, elected among the terminals. The leader terminal is the only one able to update the global state of the system by processing the events produced by the players. The **follower** terminal, and the leader too, continuously monitor the `pong_lobby` key and update their local state to match the one stored in the etcd database. Both terminals use another thread to manage **leadership**:

- the leader processes the events and keeps refreshing his leadership lease
- the follower watches over the leader key in the etcd database and campaigns for leadership as soon as the key expires

3.3 Interaction

In this part, interactions between the main entities of the system will be shown using sequence diagrams.

3.3.1 Centralized architecture

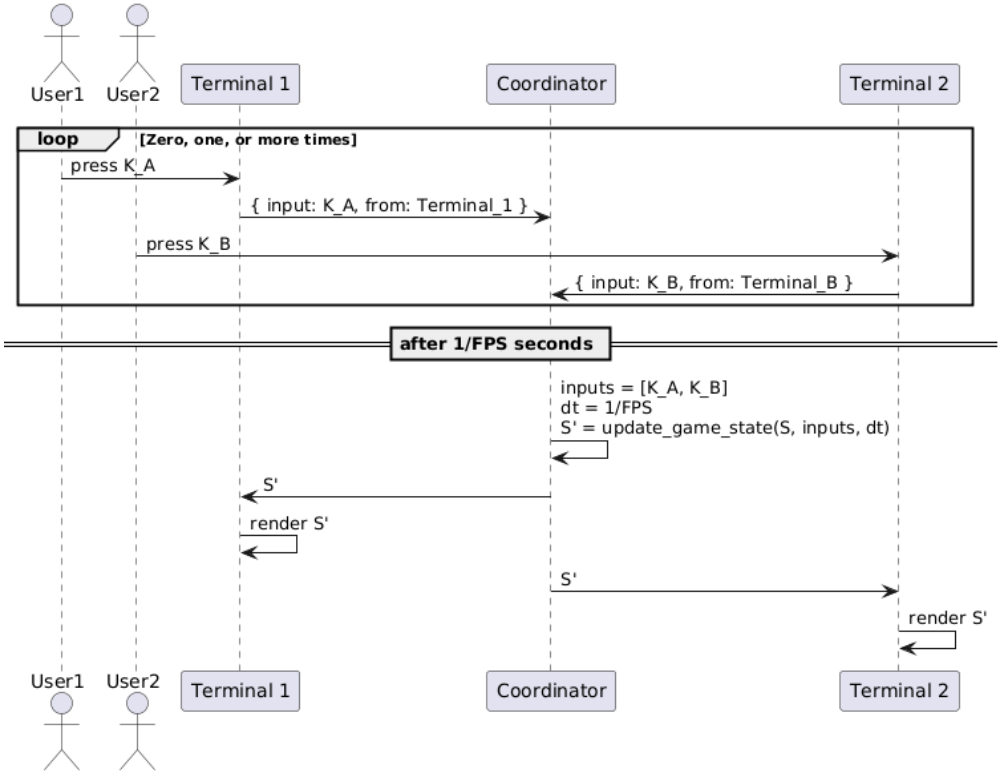


Figure 12: The interactions between `Coordinator` and `Terminals` in the `centralised` version of the system

The diagram depicts a scenario where two users interact through separate terminals managed by a `Coordinator`. Each user, operating their own `Terminal`, can press keys that represent player actions. When a user presses a key (for example, `K_A` from `Terminal 1` or `K_B` from `Terminal 2`), the `Terminal` relays the input to a `Coordinator`. The `Coordinator` collects these inputs over a brief time interval—until the next frame interval elapses. After 1/FPS seconds, the `Coordinator` aggregates the collected inputs from both terminals and updates the game state accordingly (resulting in a new state S'). Each

`Terminal` then retrieves this updated state and renders it locally. This process repeats continuously, ensuring that every user action is translated into meaningful changes in the shared game state and that each user's view of the game remains synchronized and up to date.

3.3.2 Etcd-based architecture

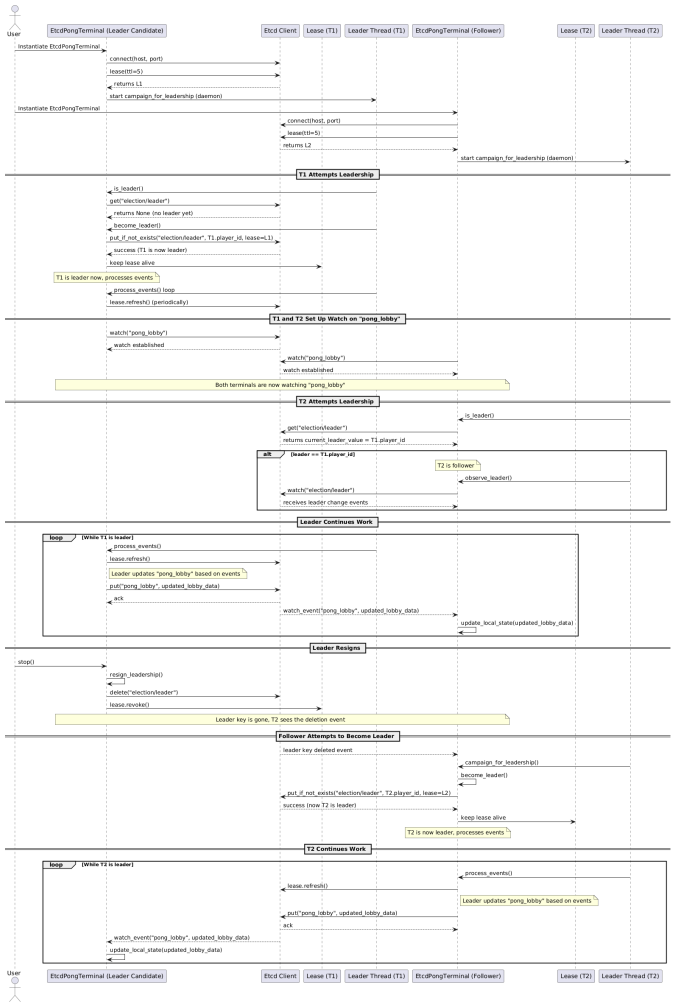


Figure 13: The interactions between two `EtcdPongTerminal` instances

The diagram illustrates the sequence of operations performed by two `EtcdPongTerminal` instances—one acting as a leader candidate and the other as a follower—when connecting to an **etcd** server and participating in a leader election process. When a terminal is instantiated, it attempts to establish a connection with the etcd server and acquire a **lease**. Subsequently, a dedicated background thread tries to become the leader by creating a key in the distributed store. If no leader exists, the terminal's `put_if_not_exists` call

succeeds, designating it as the leader and allowing it to process game events, refresh the lease, and effectively maintain leadership. In addition, both terminals register watches on a dedicated key—named `pong_lobby`—which contains the current state of the game (such as paddles and ball positions). As the leader processes user inputs and updates the lobby state within etcd, the follower, and the leader terminal itself, automatically receive these changes due to their watch on the `pong_lobby` key. Each terminal then updates its local state to reflect the newly retrieved state of the game, ensuring that it remains synchronized across all participants. When the leader eventually resigns, the leader key is deleted, signaling to the follower that it should attempt to become the new leader and continue processing the events and update the game data, thereby managing eventual failures with the leader.

4 Comparative Analysis of Communication Protocols

In this section, we analyze how each communication protocol influences many characteristics of the project. Each technology offers distinct tradeoffs between consistency, reliability, and availability.

4.1 UDP

UDP is characterized by its minimal overhead and fastest performance, making it the least reliable protocol in terms of consistency and fault tolerance. Key characteristics include:

- **Lowest consistency guarantees**: since it does not use **ACKs**
- **No built-in packet delivery confirmation**: leading to potential message loss or out-of-order delivery (thus there is **no consistency**)
- **High performance:** due to minimal overhead (this guarantees a strong **availability** of the system)

Tradeoff: UDP achieves the highest performance (and availability) but sacrifices consistency, which makes it suitable for games where the most important aspect is a low latency.

4.2 WebSockets

Built on TCP, WebSockets provide reliable and ordered message delivery. They are notable for their ability to support failure recovery, making them a more robust choice for applications requiring consistent communication:

- **High consistency guarantees**: WebSockets ensure reliable message delivery and maintain the order of messages since they are based on TCP
- **Higher overhead than UDP**: this result in worse availability and increased bandwidth usage

Tradeoff: WebSockets strike a balance between reliability and performance, making them suitable for real-time communication scenarios.

4.3 ZeroMQ

ZeroMQ is a **message-oriented** networking library. This results in transport agnostic sockets that can rely on numerous messaging patterns. This is quite useful to manage the topology of the network of our distributed system.

Key characteristics include:

- **Many Transport Options:** ZeroMQ works with both TCP and UDP, offering users the ability to pick a specific protocol based on their use case
- **Versatile Communication Patterns:** Built-in templates like publish-subscribe, request-reply, and pipeline make it easy to design robust systems with minimal effort.
- **Built for Scale:** Its architecture simplifies managing distributed systems, enabling smooth operation even in dynamic, large-scale systems

Tradeoff: The performance of communication based on ZeroMQ is strongly linked to the technologies picked up in the implementation, since ZeroMQ can rely on TCP and UPD communication protocols under the hood.

4.4 etcd

Etcd stands out for its strong fault tolerance and reliability, achieved through leader election, lease management, and state replication. However, these features come at the cost of slower performance due to consensus overhead. Although it offers the highest reliability, its complexity often makes it overkill for simpler applications such as distributed games, where performance is critical.

Key characteristics include:

- **Consensus-Based Consistency:** Using the Raft consensus algorithm, etcd ensures that all nodes in the cluster agree on the current state of the system.
- **Fault Tolerance:** Redundancy and state replication enable the system to remain operational despite node failures.
- **Performance Overheads:** The complexity of consensus operations results in higher latency and reduced throughput compared to other protocols.

Trade-off: Etcd prioritizes consistency over availability, making it unsuitable for latency-sensitive applications such as real-time multiplayer gaming.

5 Implementation Details

5.1 Technologies

5.1.1 ZeroMQ

ZeroMQ[6] "looks like an embeddable networking library but acts like a concurrency framework". It was used as requested as one of the alternative communication protocol. The ROUTER-DEALER pattern was implemented to facilitate flexible and asynchronous message routing between terminals and coordinators, surpassing the limitations of the traditional synchronous REQ-REP (Request-Reply) pattern. This architecture was particularly advantageous to allow non-blocking communication and dynamic message handling.

5.1.2 FastAPI

FastAPI[2] is a modern, fast (high-performance), web framework for building APIs with Python based on standard Python type hints. It was used to implement the RESTful API needed for the lobby management of the WebSockets oriented implementation of the game. Its straightforward usage made it possible to implement these functionalities quite fast and easily.

5.1.3 Uvicorn

Uvicorn[5] is an ASGI web server implementation for Python. It was used to manage the requests received on the Lobby API.

5.1.4 Pydantic

Pydantic[4] "is the most widely used data validation library for Python". In the project, it was used to manage the validation of the requests and responses used by the clients and the server in the Lobby API.

5.1.5 Etcd

Etcd[1] is a strongly consistent, distributed key-value store that provides a reliable way to store data that needs to be accessed by a distributed system or cluster of machines. It gracefully handles leader elections during network partitions and can tolerate machine failure, even in the leader node. In the case of my project, etcd was the best way to implement a decentralised version of the original centralised architecture. It allowed the easy implementation of a fault tolerant leader election.

5.2 JSON Schema

JSON Schema[3] "enables the confident and reliable use of the JSON data format". It was used to ensure consistency in the etcd implementation of the system. This way, events and lobby state could be validated and checked before being stored in the distributed store.

6 Validation

The fulfillment of the requirements was ascertained by keeping a constant dialogue with Prof. Ciatto since the requirements were added iteratively while the work progressed. No particular validation strategy was used to formarly validate the system since the nature of the implementations was hard to formalize in detailed tests.

7 Deployment Instructions

7.1 Centralised architecture

To run the centralised version of the project it's needed to follow these steps:

- Install **Poetry**. Poetry can be installed using `pipx`, which is a tool for installing Python CLI applications globally while isolating them in virtual environments. Execute the following command:

  ```
  pipx install poetry
  ```

  For other installation methods, refer to the official documentation: `https://python-poetry.org/docs/#installing-with-pipx`.

- Install the project's dependencies by running:

  ```
  poetry install
  ```

- Activate the **Poetry shell** by executing:

  ```
  poetry shell
  ```

- Depending on the communication protocol, execute one of the following commands:
  - For the coordinator role:

    ```
    python -m dpongpy --mode centralised --role coordinator --host localhost
    --port 5000 --comm-type COMM_PROTOCOL
    ```

  - For the terminal role on the right side:

    ```
    python -m dpongpy --mode centralised --role terminal --host localhost
    --port 5000 --side right --keys arrows --comm-type COMM_PROTOCOL
    ```

  - For the terminal role on the left side:

    ```
    python -m dpongpy --mode centralised --role terminal --host localhost
    --port 5000 --side left --keys arrows --comm-type COMM_PROTOCOL
    ```

- In these commands, replace `COMM_PROTOCOL` with one of the following supported communication protocols:
  - `udp`
  - `zmq`
  - `web_sockets`

7.2 Etcd-based architecture

To run the etcd-based architecture, simply execute the provided `docker-compose.yml` file with the following commands:

- Build the Docker containers:

  ```
  docker-compose build
  ```

- Start the Docker containers:

  ```
  docker-compose up
  ```

8 Usage Examples

Since the usages are pretty similar to each other, the implementation based on **WebSockets** will be shown since more meaningful.

After starting the `Coordinator`, the server will be ready to receive requests on the Lobby API:

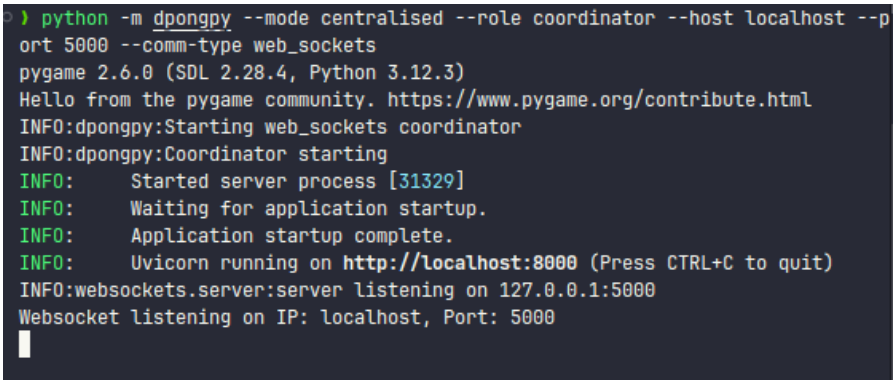


Figure 14: Coordinator waiting for terminals to join

After one `Terminal` joins, the `Coordinator` will wait for the remaining players:

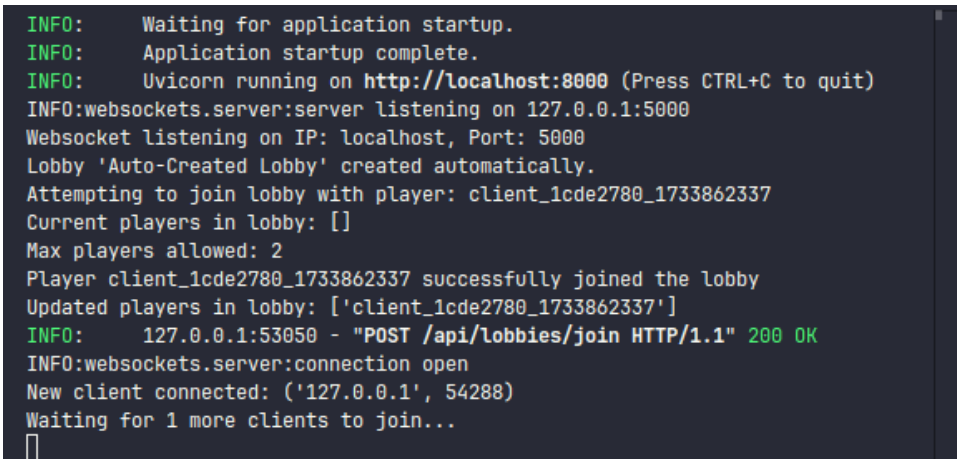


Figure 15: `Coordinator` waiting for another `Terminal`

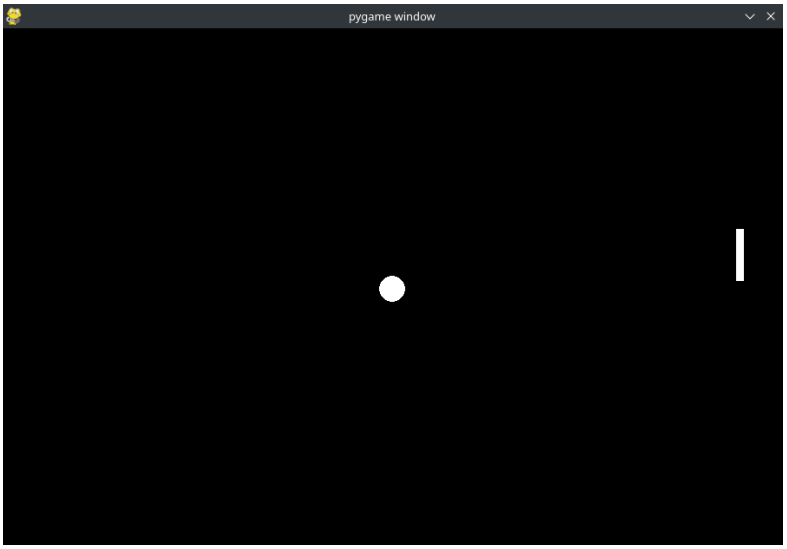


Figure 16: The `Terminal` view of the game

Then, after the other `Terminal` joins, the game starts and the ball starts moving:

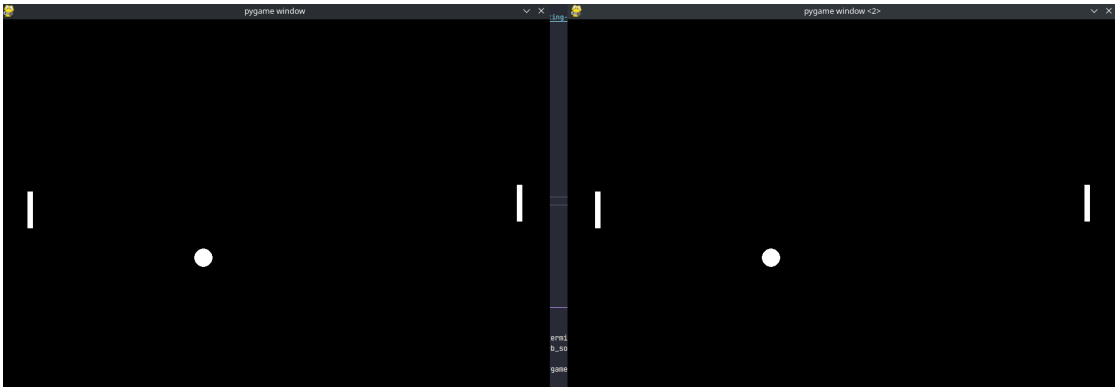


Figure 17: The two terminals displaying the same state of the system while the ball moves

9 Conclusions

This project aimed to extend the Distributed Pong game with additional communication protocols and a decentralised architecture for educative purposes.
The project was successful in implementing ZeroMQ and WebSockets as alternative communication protocols. The etcd-based implementation provided a decentralised alternative to the centralised coordinator-terminal pattern, showcasing a different approach to distributed game state management. The system's architecture was designed to be flexible and modular, allowing for easy integration of new communication protocols and

state management solutions. The project's codebase was structured to be clear and well-documented, serving as an educational example of distributed system concepts and possible implementations of several networking protocols.

9.1 Future Works

In future works more communication protocols could be implemented. Also improving the performance and reactivity of the etcd-based implementation would be a good follow-up since in this project it was just implemented as educational showcase of the capabilities of etcd.

9.2 What I've learned

This project was a good experience to improve my skills in addressing several issues and features of distributed systems. Personally, the most instructive part of this work was to work on an unfamiliar codebase and extend its features. This, together with the many issues that I encountered while working, made this project challenging but quite fulfilling to do.

References

[1] Etcd. `https://etcd.io/`.

[2] FastAPI. `https://fastapi.tiangolo.com/`.

[3] JSON Schema. `https://json-schema.org/`.

[4] Pydantic. `https://docs.pydantic.dev/latest/`.

[5] Uvicorn. `https://www.uvicorn.org/`.

[6] ZeroMQ. `https://zeromq.org/`.