Final Report: DS-Cinema

Distributed P2P Seat Reservation System

Final Report for the Distributed Systems Course 2025-2026

Luca Cantagallo
luca.cantagallo@studio.unibo.it

January 29, 2026

DS-Cinema is a distributed peer-to-peer system for cinema seat reservation. There is no central server managing the application logic or storing data: everything is distributed across the nodes in the network. The goal of the project is to implement from scratch the distributed coordination algorithms studied in class. In particular, I used the Ricart-Agrawala algorithm together with Lamport Clocks to guarantee **mutual exclusion**: if two users click at the same instant, they won't have problems, but one of them will get the seat as booked, while for the other it will appear as occupied. The system also handles failures: if a node crashes during a reservation, the others detect it via timeouts and continue to work normally. New nodes can dynamically join the network and synchronize with the current state.

1 Concept

The project is a distributed peer-to-peer application for managing seat reservations in a cinema hall. Unlike traditional client-server architectures, where a central database updates the various nodes and interfaces with them, in DS-Cinema the nodes themselves manage the state and coordinate to maintain a consistent and up-to-date view of the hall.

What it consists of: a desktop application with a minimal graphical interface, supported by a lightweight discovery server running from the terminal to get updates on the presence of new nodes and maintain their addresses.

Use case: When opening the application, a user who connects sees, through the graphical interface, a grid of seats, which can be green or red. The red ones have been taken previously by other users, while the green ones are still available. They can book a seat by clicking on one of the still-green seats. If they manage to acquire the seat (click and it succeeds, meaning no one else has booked it at that same instant or that it won the place in that conflict), they acquire the seat and it becomes dark green, to indicate that it is theirs. There is no central database nor local persistent storage. The state

is maintained in-memory and replicated across active nodes. When a node connects or reconnects, it downloads the current state from the other active peers.

Why is distribution needed?

- **Fault Tolerance:** By eliminating the central application server, there is no longer a single point of failure for the business logic
- **Resource Sharing:** Multiple independent users compete for a finite number of exclusive resources (the seats)
- **Educational purpose:** To implement distributed coordination algorithms (mutual exclusion, logical time) without relying on high-level frameworks

2 Requirements

I designed the system to satisfy the following requirements, prioritizing the consistency of the distributed state.

Functional Requirements:

1. **Visualization:** Each node must display the real-time status of the seats (Free, Mine, Occupied, Pending)
2. **Booking:** A user must be able to book a free seat. The system grants the reservation only if no other user has already taken it (mutual exclusion)
3. **Release:** A user can release a seat they own, and the update is propagated to all other nodes
4. **Discovery:** New nodes must be able to join the network and dynamically discover existing peers

Non-Functional Requirements:

1. **Consistency:** The system must prevent double-booking. In terms of the CAP theorem, it is a CP system: in case of network partition, consistency takes priority over availability
2. **Fault Tolerance:** The system assumes a Fail-Stop model. It must detect crashed nodes and exclude them from the coordination process to avoid deadlocks
3. **Persistence and Recovery:** A node restarting after a crash (or joining late) must be able to obtain the current system state

Implementation choices:

Python 3.10+, raw TCP Sockets (no RPC frameworks), and Test-Driven Development (TDD) following GitHub Flow.

2.1 Relevant Distributed System Features

- **Fault Tolerance:** High relevance. The Ricart-Agrawala algorithm requires all nodes to reply to guarantee mutual exclusion. The system must be able to detect crashed nodes and exclude them from the coordination process, otherwise there is a risk of indefinite blocking (liveness violation).
- **Concurrency & Synchronization:** High relevance. The main use case involves multiple users potentially clicking on the same seat at the same instant (for example, when booking opens). The system must resolve these conflicts deterministically and ensure that only one user gets the seat.
- **Scalability:** Low relevance. The realistic use case is a small cinema hall with 3-10 users connected simultaneously. There is no need to optimize for hundreds of nodes, so algorithms with O(N) complexity are acceptable.
- **Transparency:** Partial relevance. The user should not have to worry about network details (who the other nodes are, how they communicate), but it is acceptable to visually show when a distributed negotiation is ongoing, to give feedback that the request is being processed.

3 Design

I designed the system following a decentralized approach, where the logic is not on a central server but is distributed directly on the nodes (peers).

3.1 Architecture

The system uses a **Hybrid Peer-to-Peer** architecture. The idea is this: all the main business logic (seat reservation and critical section management) is decentralized among peers. This ensures there is no single point of failure: if a node crashes, the others continue to work normally.

However, there is a practical problem: when a node starts up, how does it know who the other nodes in the network are? To solve this I used a lightweight `NameServer`, which only serves for the initial discovery phase. It is important to emphasize that this server does not manage any booking logic and does not store the seat state: it only serves as an "address book" to let nodes meet at the beginning.

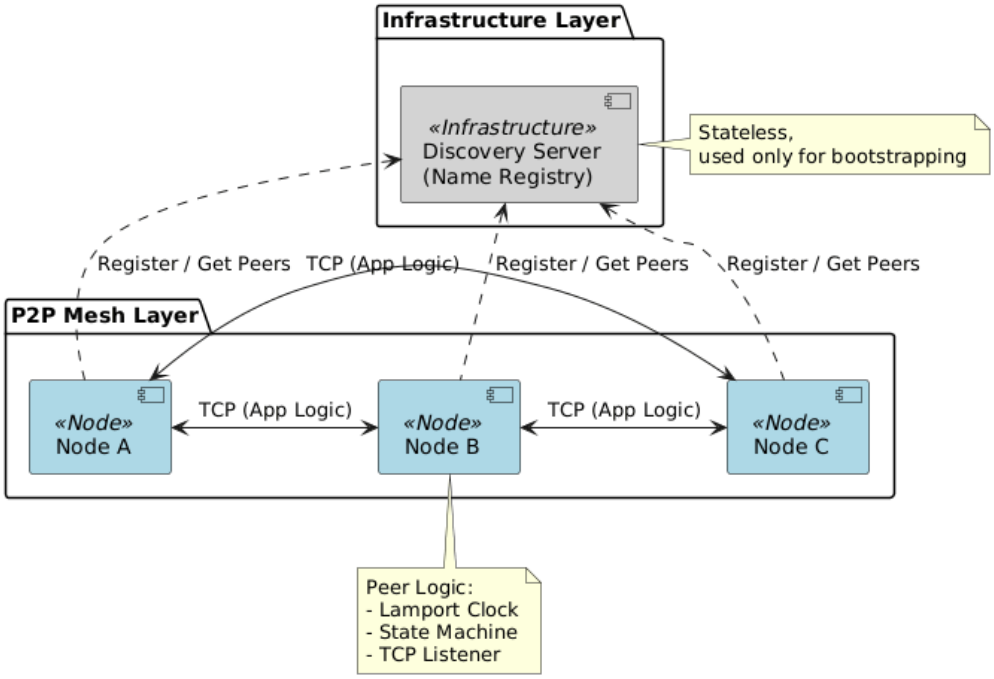

Figure 1: Hybrid P2P Architecture: A central Discovery Server aids bootstrapping, while logic is handled via a P2P Mesh.

3.1.1 Node Design

Each node must do two things simultaneously: act as a client (when it wants to book a seat and needs to communicate with others) and as a server (when it needs to receive requests from other nodes).

Network operations can be slow and block the user interface. To avoid this, I used a multi-threaded architecture:

- **Main Thread:** Manages the application state and the GUI event loop. It never blocks for netwokr operations.
- **Network Thread:** Runs in the background listening for incoming TCP connections. When a connection arrives, it spawns a short-lived worker thread that handles that specific request. This way network latency does not affect interface responsiveness.

3.1.2 Communication Protocol

For communication between nodes I used raw TCP sockets, without high-level frameworks. This gave me complete control over the protocol, but also created an interesting technical challenge: TCP ”sticky packets”.

The problem is this: TCP is a stream protocol, not a message protocol. When you send two consecutive JSON messages, they might arrive fused together in a single block of bytes. The receiver gets a buffer like `{"msg":1}{"msg":2}` and must figure out where the first message ends and the second begins.

To solve this I implemented a custom application-layer framing protocol:

- **Message Framing:** Each message is preceded by a 4-byte header containing a Big-Endian integer. This integer indicates how many bytes the following JSON payload occupies. When the receiver reads the first 4 bytes, it knows exactly how many more bytes it needs to read to have the complete message. After processing it, it can check if there are other messages in the buffer and repeat the process.
- **Serialization:** I chose JSON (UTF-8) for the payload. A binary format would have been more efficient, but for this project readability during debugging was more important.

Byte 0	**Byte 1**	**Byte 2**	**Byte 3**	**Bytes 4...N**
Payload Length (Big-Endian Integer)				*JSON Payload* (UTF-8)

Table 1: Structure of the custom Application-Layer Framing Protocol designed to solve TCP sticky packets.

3.2 Modelling

Domain Entities:

I modeled two main entities:

- **Seat:** Identified by a numeric ID (from 0 to 24 in the 5x5 grid). Each seat can be in three states:
  - FREE: The seat is available, no one has booked it
  - PENDING: A negotiation is ongoing, the system is deciding who to assign it to
  - BOOKED: The seat has been booked by someone
- **Message:** The basic unit of communication between nodes. I defined several message types:
  - `REQUEST`: When a node wants to enter the critical section (book a seat)
  - `REPLY`: Response to a REQUEST, giving permission to proceed
  - `RELEASE`: Communication that a seat has been booked or released
  - `SYNC`: Peer list update from the NameServer

– `STATE_REQUEST`: Request for the complete state from a node that is joining

State Management:

Each peer maintains locally, in memory, a dictionary that maps each seat ID to its owner. If a seat is free, the value is `None`, otherwise it contains the ID of the node that owns it.

An important choice I made was not to use the local disk for persistence. Initially I had thought of saving the state to a JSON file, but this creates a problem: if a node restarts and reads its local file, that state might be old (stale). In the meantime others might have booked or released seats.

Instead I implemented a **Network State Transfer**: when a node joins (or reconnects after a crash), it does not read from disk but asks the other peers directly for the current state. This guarantees that it always has the most up-to-date version.

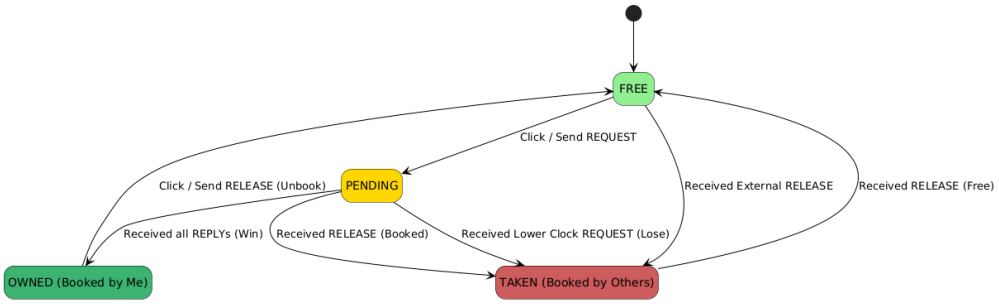

Figure 2: State Machine Diagram illustrating the logical lifecycle of a seat. Transitions are triggered by user actions or network messages.

3.3 Interaction (The Algorithm)

This is the most important part of the project. To ensure mutual exclusion, meaning that two users cannot book the same seat, I implemented the **Ricart-Agrawala algorithm** combined with **Lamport Logical Clocks**.

The algorithm works in three phases:

1. Request Phase:

Imagine that user A clicks on a seat. Here's what happens:

- Node A increments its local Lamport Clock. This clock is a counter that keeps track of events. Every time something significant happens (like a booking request), the clock is incremented.
- Node A creates a `REQUEST` message containing three pieces of information: the current value of its clock (let's call it C_A), its ID (ID_A), and which seat it wants to book (SeatID).
- This message is broadcast to all other peers the node knows.

- Locally, the seat changes state from FREE to PENDING, and in the GUI it becomes yellow. This gives visual feedback to the user that the request is in progress.

2. Contention Phase:
Now let's put ourselves in the shoes of node B that receives this REQUEST. Node B must decide what to do, and there are two cases:

- **Simple case:** If B is not interested in that seat (maybe it didn't even click, or clicked on another seat), then it immediately responds with a `REPLY` message. It's like saying "ok, go ahead, I'm not interested".
- **Conflict case:** If instead B is also trying to book the same seat, there is contention. Both want the same seat and sent REQUESTs more or less at the same time. Who wins?

Here the **Tie-Breaker Rule** based on total ordering comes into play. The rule says that request A has priority over request B (we write $A \Rightarrow B$) if one of these two conditions is true:

$$C(e_A) < C(e_B) \quad \vee \quad (C(e_A) == C(e_B) \wedge ID_A < ID_B) \tag{1}$$

In simple words: whoever has the lower timestamp wins. If by chance the timestamps are identical (the two nodes clicked at exactly the same logical instant), then we use the node ID as a tie-breaker: whoever has the lexicographically smaller ID wins (for example "Luca" beats "Marco").
So if A wins according to this rule:

- B immediately sends `REPLY` to A (gives it permission)
- B pauses its request and waits

If instead B wins:

- B does NOT send REPLY to A
- B "defers" the response: it will put it in a queue and send it only after completing its critical section

This mechanism ensures there are no deadlocks: there is always a deterministic ordering, so it cannot happen that A waits for B and B waits for A simultaneously.
3. Critical Section:
Node A remains waiting until it receives `REPLY` from all live nodes. As soon as it has all the necessary responses, it can finally enter the critical section:

- The seat is booked: locally it becomes BOOKED
- In the GUI, the seat becomes dark green for A (because it's theirs), while for other nodes it will become red (because it's occupied)

- A broadcasts a `SEAT_TAKEN` message to inform everyone else of the state change. This message does not require permission: it's just a notification

After a brief delay (to simulate an operation in the critical section), the node exits the critical section and sends REPLY to all those it had deferred.

3.3.1 Complexity Analysis

How many messages are needed for a single booking? Let's count:

In the best case, without failures:

- Node A sends **N - 1** REQUEST messages (one to every other node, excluding itself)
- Every other node responds with REPLY, so A receives **N - 1** REPLY messages

In total that's **2(N - 1)** messages for each access to the critical section. This means **O(N)** complexity, linear in the number of nodes.

For my use case (2-10 nodes), this complexity is perfectly acceptable. With 10 nodes it's 18 messages per booking, which is negligible. However, if I had to scale to hundreds of nodes, it would become a problem: with 100 nodes it would be 198 messages per booking. In that case it would be better to use more scalable algorithms like Raymond's (which uses a tree structure and has O(log N) complexity), but that's out of scope for this project.

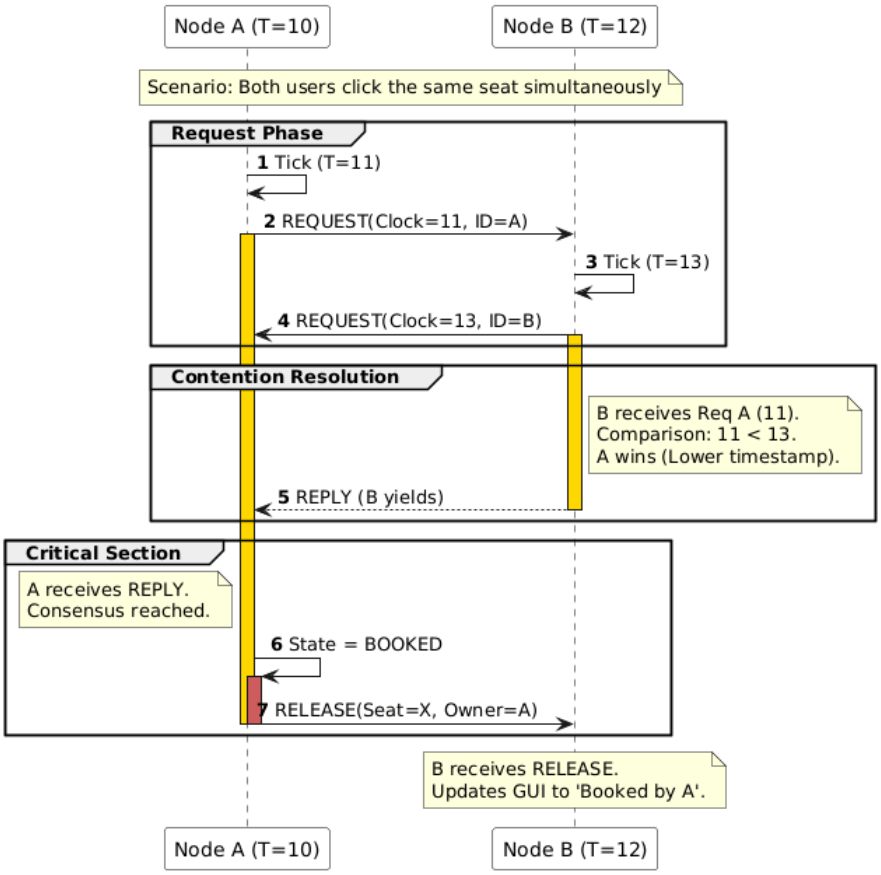

Figure 3: Sequence Diagram: Node A ($T = 10$) wins against Node B ($T = 12$) because its timestamp is lower.

3.4 Data and Consistency

Storage:

As I said before, the state is completely in memory. There is no disk persistence, everything is replicated among active nodes through the network.

Consistency Model:

The system guarantees **Sequential Consistency** for each individual seat. What does this mean? It means that all nodes will see the operations on a particular seat in the same order.

This is possible thanks to Lamport Clocks which provide total ordering of events: if A books before B according to logical clocks, all nodes will agree on this order. It cannot happen that for node C it seems B booked before A while for node D it's the opposite.

3.5 Fault Tolerance

The system handles **Crash Failures**, following the Fail-Stop model. That is, I assume that when a node crashes, it stops completely and does not send corrupted or inconsistent messages.

Mechanism:

I implemented active timeouts. It works like this:

When node A sends REQUEST broadcasts, it starts a timer for each destination. If a peer does not respond within T seconds (I set $T = 2.0$ seconds), A concludes that node has crashed.

Behavior:

As soon as A detects a crash, it does two things:

1. Removes the dead node from its local directory (so it won't send it more messages in the future)
2. Calls a callback that **immediately re-evaluates** whether it can enter the critical section

Why immediate re-evaluation? Imagine this scenario: there are 4 nodes (A, B, C, D). A wants to book a seat and sends REQUEST to all. B and C respond with REPLY, but D has crashed. Without re-evaluation, A would remain blocked waiting for D forever (liveness violation).

With re-evaluation, A realizes D is dead, removes it from the list, and realizes it already has all the necessary REPLYs from the new quorum (only B and C). At this point it can immediately enter the critical section without further delays.

This mechanism guarantees **Liveness**: the system continues to progress even in the presence of failures, as long as there is at least one functioning node.

4 Implementation

As I mentioned in previous sections, I implemented the entire project in **Python 3.10+**.

4.1 Technological Details

Networking (Socket):

For communication between nodes I used Python's `socket` module, which allows working directly with raw TCP sockets. This choice gave me complete control over the protocol, but it also meant having to manually handle many low-level details.

The main problem was handling TCP "sticky packets". As I explained in the Design section, TCP is a stream protocol and does not guarantee that messages arrive separated. I had to design a buffering mechanism that reads the first 4 bytes to know how long the message is, then reads exactly that amount of bytes, and finally checks if there are other messages in the buffer to process.

Concurrency (Threading):

The `threading` module is fundamental to the node's architecture. It allows me to separate blocking I/O operations (like waiting for a TCP connection) from the GUI event loop. Without threading, every time the node waits for a network response, the entire interface would freeze.

An important aspect is synchronization: to avoid race conditions, I used **Reentrant Lock** (`threading.RLock`). I chose RLock instead of a normal Lock because it allows the same thread to acquire the lock multiple times without blocking (useful when you have recursive calls between the network layer and the logic layer). This prevents deadlocks in complex scenarios where a function that already has the lock calls another function that tries to reacquire it.

GUI (Tkinter):

The graphical interface is built with `tkinter`, Python's standard GUI library. Tkinter has an important constraint: everything that modifies the GUI must be executed on the main thread.

To respect this constraint while keeping the interface responsive, I structured the code like this: when a message arrives from the network (on a worker thread), I don't directly modify the GUI widgets. Instead, I use the `.after()` method to schedule the update on the main thread. Basically `.after(0, lambda: update_function())` tells Tkinter "execute this function as soon as possible on the main thread". This guarantees thread safety and prevents crashes or strange GUI behaviors.

4.1.1 Core Algorithm Implementation

This code snippet shows the central part of the algorithm: the request to enter the critical section. It's an important piece because it must be atomic and thread-safe.

Let's see what happens:

1. I acquire the lock to ensure that no other thread can modify the state while I'm doing this operation
2. I check if I'm already in WANTED or HELD state: if yes, I exit immediately (I can't make two simultaneous requests)
3. I change the local state to WANTED
4. I increment the Lamport Clock following the local event rule
5. I save the timestamp of my request
6. I prepare the REQUEST message with all the necessary information
7. **Crucial:** I release the lock BEFORE doing the broadcast. Why? Because the broadcast is a network operation that can take time, and if I kept the lock blocked during the broadcast, I would slow down the whole system. I/O operations should always stay outside critical sections when possible

Listing 1: Ricart-Agrawala Request Logic

```
def request_critical_section(self):
    with self._lock:
        if self.state != State.FREE:
            return False

        self.state = State.WANTED

        self.clock.increment()
        self.request_ts = self.clock.value

        msg = {
            "type": MessageType.REQUEST,
            "sender_id": self.node_id,
            "timestamp": self.request_ts
        }

    self.network.broadcast(msg)
    return True
```

This pattern (lock → modify state → unlock → I/O) guarantees the *safety* property of mutual exclusion: the state is modified atomically, but the network is not unnecessarily blocked.

4.2 Test-Driven Development (TDD)

As suggested in the course guidelines, I adopted a TDD approach.

Before implementing all the distributed network logic (which is complex and difficult to debug), I wrote unit tests using `pytest` for the fundamental components:

Lamport Clock: I wrote tests to verify that the clock increments correctly on local events, and that the update works as expected when a message is received (it must do `max(local, remote) + 1`). I also tested concurrent event scenarios to make sure total ordering worked correctly.

Protocol Serialization: I tested that JSON messages are correctly packed into the byte stream (header + payload) and that deserialization can correctly extract messages even when there are multiple in the buffer or when they arrive fragmented.

This approach saved me a lot of time. When I then integrated everything and started testing the actual network communication, I already knew that the protocol and clocks were working correctly. So if something went wrong, I knew the problem was in the distributed coordination logic, not in the base components. This made debugging much more manageable.

5 Validation

I validated the system on two levels: automated unit tests for critical components and manual end-to-end tests to verify distributed behavior.

5.1 Automated Testing

As I explained in the Implementation section, I followed a TDD approach. I wrote unit tests using `pytest` to validate the fundamental components before integrating them:

- **Lamport Clock:** I tested that the clock increments correctly (+1) on each local event, and that the update works as expected when receiving a message: it must calculate $max(L_{local}, L_{remote}) + 1$. I also verified scenarios with concurrent events to ensure that total ordering was deterministic.
- **Protocol:** I tested message serialization and deserialization. In particular, I verified that the 4-byte header is correctly written and read, and that the parser can correctly handle complex situations like multiple messages in the same buffer (sticky packets) or fragmented messages arriving in multiple chunks.
- **NameServer:** I tested the logic for peer registration, update, and removal from the directory.
- **Race Simulation:** I implemented a test that simulates simultaneous clicks using a MockTransport to verify that tie-breaking works correctly in case of identical timestamps.

5.2 Manual Acceptance Testing

After validating the base components, I manually tested the complete system in realistic distributed scenarios.

5.2.1 End-to-End Test with Multiple Sessions

I performed a complete test simulating a real use case with three users (Luca, Marco, Valerio):

1. I start the NameServer (via Docker)
2. Luca enters first: he sees all 25 seats in green (available). He books seats 14 and 15, which become dark green in his GUI
3. Marco joins the cluster: his GUI immediately shows seats 14 and 15 in red (occupied by Luca). This confirms that the State Transfer mechanism works correctly
4. Valerio joins: he also sees the same 2 seats in red
5. Marco books seat 8: it becomes dark green for him. Simultaneously, both Luca and Valerio see their GUIs update and seat 8 becomes red. This demonstrates that broadcast works and GUIs synchronize in real-time
6. Valerio exits the cluster

7. Luca releases one of his bookings: Marco immediately sees the seat turn green in his GUI
8. Marco books the just-released seat: it becomes dark green for him
9. Marco exits the cluster
10. Luca books another seat without problems, then exits

This test demonstrates that the system correctly handles:

- Dynamic join of new nodes with State Transfer
- Real-time propagation of bookings and releases
- Automatic GUI synchronization

6 Deployment

In this section I describe how to prepare the environment to run the system locally. The main steps are three: clone the repository, install dependencies for the nodes, start the NameServer with Docker.

6.1 Clone the project

To obtain the source code:

```
git clone https://github.com/LucaCantagallo/DS-Cinema.git
cd DS-Cinema
```

6.2 Install dependencies for nodes

The application nodes run on the host machine. To manage the Python environment I use `poetry`. After cloning, it is sufficient to execute once:

```
poetry install
```

This command prepares the virtual environment and installs the dependencies defined in `pyproject.toml`. Subsequently, nodes will be executed via `poetry run`, as described in the User Guide.

6.3 NameServer with Docker

The `NameServer` runs inside a Docker container defined in the `docker-compose.yml` file. The configuration exposes port `5000` and sets the necessary variables so that the NameServer can communicate with nodes on the host.

Starting requires a single command:

```
docker compose up nameserver
```

Once the service is started, the system is ready for launching the application nodes described in section 7.

7 User Guide

In this section I describe how to start the system and how to interact with the GUI from a user's perspective.

7.1 Starting the system

1. **Start the NameServer** (from a first shell):

   ```
   docker compose up nameserver
   ```

   Wait for the log indicating that the NameServer is listening on port 5000.

2. **Start the application nodes** (from a different shell for each node):

   ```
   poetry run python -m src.node.main Luca 5001
   poetry run python -m src.node.main Marco 5002
   poetry run python -m src.node.main Valerio 5003
   ```

   Each command opens a new GUI window with the title `DS-Cinema Node: <NodeID>`.

7.2 Graphical interface

Each node displays:

- A 5x5 grid of buttons, one for each seat (from S0 to S24)
- A log area at the bottom showing relevant events (bookings, releases, state sync, NameServer connection errors, etc.)

The color legend is as follows:

- **Light green**: free seat, bookable

- **Dark green**: seat booked by the current node
- **Red**: seat booked by another node
- **Yellow**: *Pending* state, a distributed negotiation is in progress for that seat

Figure 4: Screenshot of three nodes (Luca, Marco, Valerio) running simultaneously.

7.3 Typical usage flow

A typical usage scenario is as follows:

1. The user starts their node and sees the updated seat map (if other nodes were already active, the state is automatically synchronized via `STATE_REQUEST/STATE_REPLY`).
2. To book a free seat, they click on a light green button:
   - The seat immediately turns yellow (Pending) while the node negotiates with other peers using Ricart-Agrawala.
   - If the booking succeeds, the seat becomes dark green on that node and red on the others.
   - If another node wins the contention, the seat returns to the color consistent with the global state (typically red).
3. If the user clicks on a seat they have already booked (dark green), the system interprets the action as a release request:
   - The seat enters Pending state (yellow) while the critical section is requested.
   - Once released, it returns to light green on all nodes.
4. If the user clicks on a red seat, the GUI shows a message in the log indicating that the seat is owned by another node and cannot be modified.

7.4 Shutting down the system

To close the system:

- Individual nodes can be closed from the X of the GUI window or with `Ctrl+C` in the terminal from which they were started.
- The NameServer can be stopped with `Ctrl+C` in the `docker compose` shell or using:

```
docker compose down
```

8 Self-evaluation

8.1 Luca Cantagallo

Role: Being an individual project, I was responsible for the entire implementation: from the low-level TCP protocol design to the integration of the distributed algorithm and GUI.

Deviation from initial proposal: Initially I had planned to implement persistence via local files (JSON). During development I decided to switch to a **Network State Transfer** approach. This choice was motivated by the fact that synchronizing state directly between peers guarantees stronger consistency and prevents "stale data" problems that are common when using local storage in distributed environments. In this way I satisfied the recovery requirement without having to handle disk I/O operations.

Strengths:

- **Practical learning of theoretical concepts:** Working with such low-level concepts (raw TCP sockets, sticky packets, manual serialization, Lamport Clock, mutual exclusion algorithms) allowed me to truly understand in depth topics that remained more abstract in class. Implementing the protocol and Ricart-Agrawala algorithm from scratch taught me in practice what it means to coordinate distributed nodes without a central server.
- **Test-Driven Development (TDD):** This was the first time I seriously applied TDD, and I discovered a concrete utility that I previously underestimated. Writing tests first for critical components like the Lamport Clock and the protocol allowed me to build the system feature by feature, testing and verifying that each piece worked correctly before moving to the next. Without TDD, debugging distributed synchronization problems would have been a nightmare. I will definitely reuse this approach in future projects.
- **Conscious use of Git:** TDD integrated very well with repository management. I tried to maintain a clean and meaningful commit history: each commit introduced new and complete features, not random changes. Especially the first commits,

when it was still unclear how to test distributed functionalities, were accompanied by their related tests. This incremental approach helped me build the system step by step, with the confidence that each step worked before adding the next one. Having atomic and tested commits was fundamental to being able to go back if something broke.

Weaknesses:

- **Graphical interface and containerization:** I had difficulties with the integration between GUI and Docker. Initially I wanted to fully containerize the application nodes as well, but Tkinter requires a graphical display and managing X11 forwarding with Docker turned out to be too complicated and not very portable. In the end I had to keep the nodes on the host and only the NameServer in Docker, which makes deployment less clean. If I were to redo the project, I would probably opt for a web-based interface that runs in the browser. This way it would be much simpler to containerize everything and have a deployment completely based on `docker compose`.
- **Scalability:** The Ricart-Agrawala algorithm has $O(N)$ message complexity for each critical section request. This works well for small groups (2-10 nodes), but would become a bottleneck with hundreds of nodes. A tree-based algorithm like Raymond's could improve scalability, but would have added complexity that was not justified for the realistic use case (a small cinema hall).

9 Future works

Possible future improvements would focus primarily on two aspects that emerged during development:

- **Web-based interface:** Replace Tkinter with an interface that runs in the browser to allow complete containerization of the system and simplify deployment.
- **Elimination of the single point of failure:** Find an alternative mechanism for the discovery phase that completely eliminates the dependency on the NameServer during bootstrap, making the system entirely decentralized.