Alma Mater Studiorum – Università di Bologna

Department of Computer Science and Engineering (DISI)

Course: Distributed Systems

Distributed Collaborative Storytelling

Final Report [2024-2025]

Gabriele ARCESE
gabriele.arcese@studio.unibo.it

Marco COSTANTINI
marco.costantini7@studio.unibo.it

February 8, 2026

Abstract

Context: We propose the development of a *Distributed Collaborative Storytelling Engine*, a real-time platform where multiple users co-author a narrative under the guidance of a rotating narrator.

Methodology: The system architecture implements a **High Availability Master-Slave cluster**. Key features include a dynamic role-rotation mechanism for the narrator and automated theme generation to ensure narrative flow. Robustness is guaranteed through a dual-layer strategy: **active state replication** for immediate failover and persistent storage for data durability.

AI Disclaimer

During the preparation of this work, the authors used GitHub Copilot to assist with code completion and documentation generation. After using this tool/service, the authors reviewed and edited the content as needed and take full responsibility for the content of the final report/artifact.

Contents

1 Concept

1.1 Type of product

The project is a networked real-time collaborative application designed to enable multiple distributed users to co-author a narrative simultaneously. Built upon a Centralized Client-Server Architecture, the system is composed of:

- **A Central Server**: Acts as the authoritative coordinator, responsible for managing the global game state, synchronizing client updates, and handling role.
- **Distributed Clients**: Lightweight desktop applications (implemented with a Python/Tkinter GUI and command line) that allow users to submit text proposals and visualize the evolving story in real-time.

The product functions as a synchronous groupware system that enforces controlled concurrency : while multiple users submit proposals in parallel, a designated Narrator acts as a serialization point, resolving conflicts by selecting the official continuation of the story for each segment.

1.2 Use case description

What the software does

The system allows a group of distributed users to collaboratively write a story. Users can propose sentences concurrently, and a designated narrator for each story segment is responsible for selecting which proposal becomes the official continuation. At the end of each story the player choose if they want to continue playing or stop. If they chose to continue the system:

- Proposes a new theme
- Rotates the narrator role to the next user
- Starts a new collaborative writing round

Where are the users

Users may be:

- On the same local network (e.g., classroom environment)
- Geographically distributed and connected through the internet
- No physical proximity is required.

When and how frequently do they interact

Interaction is real-time:

- Users can write proposals when they are in the *editing* phase within a story segment.
- The narrator validates one proposal at the end of each segment.
- Frequent communication with the server is expected (message passing, update broadcasting)

How do users interact with the system

Users interact through:

- A CLI client (or minimal GUI) that lets them:
  - Read the shared story as it evolves
  - Submit a segment proposal
  - Receive information about selected segments and the theme selected for the story

Users access the system using common devices: PCs or laptops

Roles

Player can has this roles:

- **Writer**: Can propose segments concurrently, reads updates of the main story, waits for narrator's decision.
- **Narrator**: Changes at the end of every story, selects the winning proposal among concurrent edits, ensures narrative coherence, decide, at the end of each segment, to continue or ending the story.
- **Leader**: decide when the game start, he is usually the first to join the lobby, it is a sub-role the leader can be both a writer and a narrator.
- **Spectator**: if the user connects when the game has already started, he enters this role.

1.3 Why distribution is needed

Geographically distributed participation

Users may connect from different locations, requiring a distributed network architecture to synchronize contributions in real time.

Real-time coordination of concurrent editors

Multiple writers interacting simultaneously require distributed mechanisms for gathering concurrent proposals and broadcasting state updates. Conflict resolution is handled via a centralized arbitration pattern where a designated Narrator selects the winning branch. A single local instance would not allow true concurrent editing.

Shared global state with consistency concerns

The distributed nature introduces challenges such as causal consistency, managing concurrent proposals, ensuring all clients see the same narrative evolution

Fault tolerance

The system ensures continuous operation through active connection monitoring (Heartbeat mechanism) and Server-Authoritative Timers.

Role coordination requires distributed algorithms

Narrator rotation, proposal submission, and story validation require:

- Distributed coordination
- Leader/role management
- Message passing

Which make the distributed architecture essential.

2 Requirements Elicitation and Analysis

2.1 Functional Requirements

User connection

The system must allow multiple users to connect simultaneously to the central server. **Acceptance**: at least two clients can connect and receive acknowledgment from the server.

Story visualization

Users must be able to view the shared story after the narrator decision. **Acceptance**: when the narrator validates a segment, all clients see the updated story within a bounded delay.

Concurrent proposal submission

All users can write and submit story proposals concurrently during a story segment. **Acceptance**: the server receives parallel proposals without blocking and stores them until everyone submit a proposal, after that the server send all the electible proposal to the narrator for the selection.

Narrator proposal selection

The assigned narrator must select one proposal as the official continuation for each story segment. **Acceptance**: once the narrator selects a proposal, it becomes the next segment and is broadcast to all clients.

Story segmentation

The story must progress in discrete segments, each accepting a set of concurrent proposals.

Narrator rotation

At the end of each completed story, the narrator role must be given to a random player.

Theme generation

At the beginning of each new story, the system must generate and communicate a theme.

User join in a started game

If a user join the game that is already started he is added as a spectator player and must wait the end of the round to play.

User disconnection handling

If a user disconnects, the story process must continue without interruption, the user can reconect, but if the user manage to reconnect to the game he can continue playing like nothing happened. **Acceptance**: remaining users continue normal operation after a user disconnects.

Narrator failure handling

If the narrator disconnects during a segment, the round ends and a new one begins with a different narrator.

2.2 Non-Functional Requirements

Availability

The system must support continuous interaction with minimal downtime. **Acceptance**: server uptime is up to 95% during testing.

Consistency

All users must observe the same official story order (causal consistency). **Acceptance**: no client shows a different final story sequence.

Performance

Story updates must be delivered to all clients within a reasonable delay (e.g., less than a 1 second). **Acceptance**: in testing, 95% of updates meet the latency requirement.

Scalability

The system must support at least 10 simultaneous users without performance degradation. **Acceptance**: with 10 clients connected, latency remains less than a 1 second on average.

Fault tolerance

The system must detect and recover from server failures. **Acceptance**: when server crashes, recovery occurs automatically.

Usability

The client interface must allow users to propose sentences and view updates intuitively.

2.3 Implementation Requirements

Programming language

The system must be developed in Python, following course guidelines and shared team experience.

Communication model

Server–client interaction must be based on sockets or equivalent network communication primitives, as required for distributed systems coursework.

Persistence

Story data must be stored using a lightweight storage method (e.g. JSON) chosen for ease of deployment. **Acceptance**: story logs survive server restarts.

3 Design

This chapter explains the strategies used to meet the requirements identified in the analysis.

3.1 Architecture

The system architecture implements a **High Availability (HA) Active-Passive Cluster**. While logically preserving a centralized coordinator to guarantee strong consistency and conflict resolution, the physical infrastructure is distributed across multiple server nodes.
The architecture is based on a **Master-Slave Replication Model**, where:

- The **Master Server** (Active Coordinator) maintains the authoritative global state and handles all game logic. It manages:
  - current story and active theme
  - collection of proposals and narrator assignment
  - real-time synchronization of state to Slave nodes
- The **Slave Servers** (Passive Replicas) act as *Hot Standbys*. They maintain an exact copy of the Master's in-memory state (GameState) via a dedicated replication channel but do not process client logic unless promoted.
- The **Clients** are distributed nodes that connect to the cluster. They implement client-side failover logic, automatically switching connection to a Slave node if the Master becomes unreachable.

Global Consistency & State Replication

To ensure all nodes share the same view of the story, the Master pushes state updates (JSON snapshots) to all connected Slaves via a dedicated TCP internal channel (Port 7000) immediately after any state change (e.g., a new proposal or a narrator decision). This guarantees that Slaves are always ready to take over with zero data loss in case of Master failure.

Controlled Concurrency

While proposals are submitted concurrently by users, the Master acts as the serialization point: it collects proposals, orders them, and forwards them to the narrator for final selection.

Fault Tolerance & Leader Election

The architecture replaces the traditional static server model with a dynamic **Election Mechanism**. In the event of a Master crash:

- **Detection:** Slaves detect the disconnection of the replication channel.
- **Election (Port Race):** All Slaves attempt to acquire the designated Replication Port (7000). The operating system's atomicity guarantees that only one node succeeds.
- **Promotions:** The winner is promoted to the new Master, opening its game port to clients. The losers reconnect as Slaves to the new leader.

Role Rotation and Theme Management

The Master manages narrator rotation and theme selection, ensuring fairness and consistency across sessions even after a failover event, as these attributes are part of the replicated state.

3.2 Infrastructure

Infrastructural Components

The system architecture has evolved from a single-node layout to a redundant cluster. The core elements are:

- **Server Cluster**: Composed of one active **Master** node and $N - 1$ **Slave** nodes.
  - The **Master** manages the authoritative global state, processes user proposals, and coordinates the game flow.
  - The **Slaves** operate in *Hot Standby* mode, receiving real-time state updates to ensure immediate availability in case of Master failure.
- **Multiple Client Nodes**: Distributed users connecting from different locations. They implement client-side failover logic to switch between cluster nodes transparently.
- **Replication Channel**: A dedicated, internal communication channel (TCP Port 7000) used exclusively for Master-to-Slave synchronization.
- **Persistent Storage Component**: Used by the Master to serialize completed stories and themes to **JSON files**, ensuring long-term data durability.
- **Heartbeat/Monitoring Module**: Detects node failures (both server-side and client-side) and triggers the election/failover procedures.

Distribution Across the Network

- **Server Nodes**: The cluster nodes run as independent processes, potentially on different physical machines or on different ports of the same machine for testing (e.g., Master on 65432, Slaves on 65433+).

- **Clients**: Clients are geographically distributed and independent of the server infrastructure.
- **Storage**: While state is replicated in-memory across all nodes, persistent logs are stored on the file system of the active Master.

Communication Channels

The system now employs two distinct logical channels:

1. **Public Channel (Game Protocol):** Between Clients and the Master (Game Ports).
2. **Internal Channel (Replication Protocol):** Between Master and Slaves (Port 7000) .

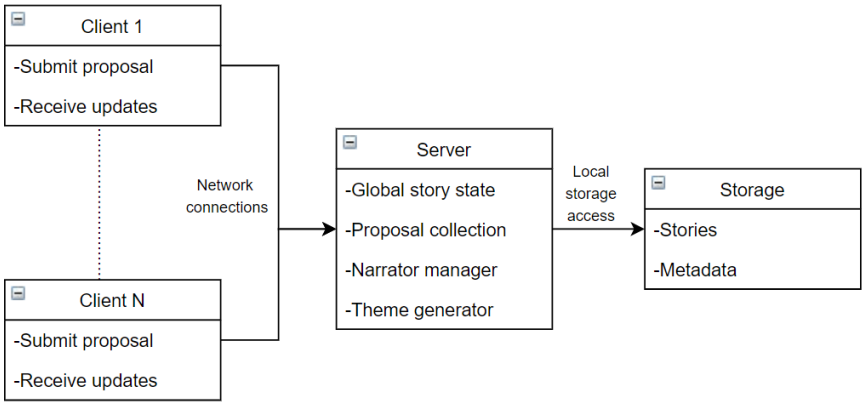

Figure 1: High Availability Cluster Architecture

Component Discovery and Naming

Unlike the previous single-endpoint model, the system uses a **List-based Discovery**. Clients are configured with a prioritized list of potential server endpoints (e.g., `[IP:65432, IP:65433, ...]`). On startup or disconnection, the client iterates through the list until an active Master is found.

3.3 Modelling

Domain Entities

The main domain entities managed by the central coordinator (`GameState`) are:

User/Player : Represents a connected client interacting with the system.

Story : The aggregate state of the narrative being constructed during a session.

Segment : Unit representing the current turn of the game loop.

Proposal : A candidate sentence submitted by a Writer for the current segment.

Voting Session : Represents the consensus mechanism triggered at the end of a game or during a reset to decide the next action.

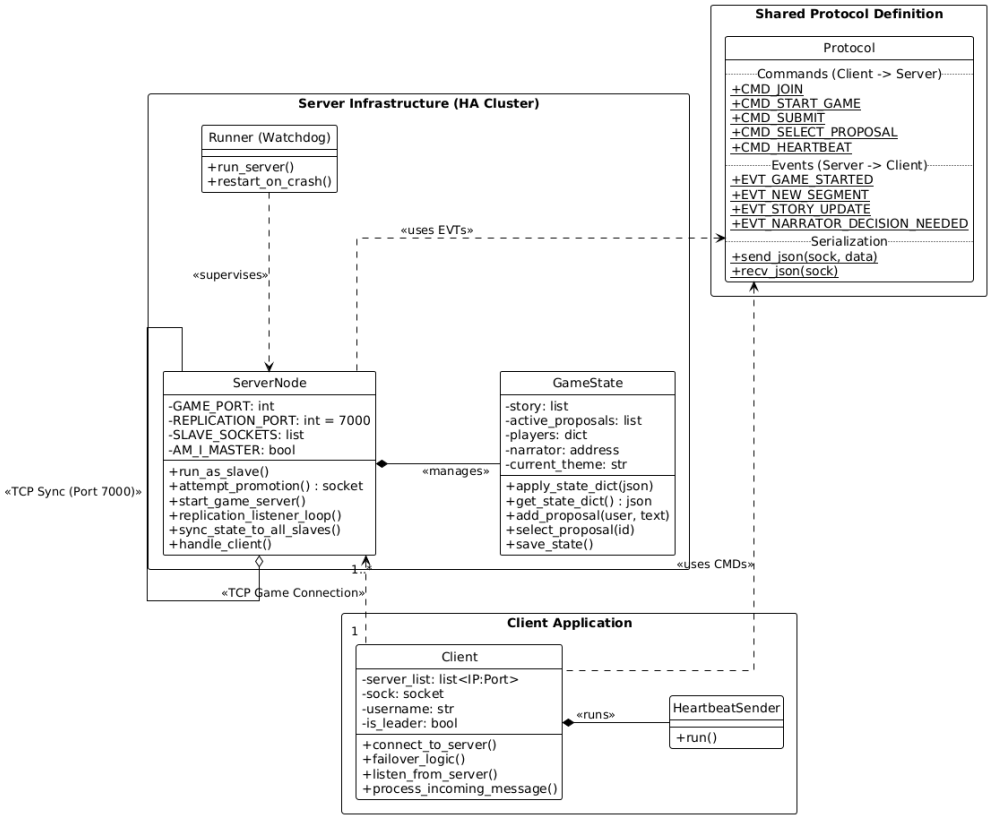

Figure 2: Diagramma classi

Domain Events

The system relies on a set of key events triggered by user actions, timer expirations, or connection state changes. These events drive the state machine of the `GameState`.

Session & Connection Management

UserJoined : A new client establishes a connection and enters the lobby. Depending on the game state, they are assigned a role as a potential player or a spectator.

UserReconnected (State Recovery) : A previously disconnected writer reconnects using the same username. The system identifies them via the whitelist, and unlocks their interface if pending actions are required.

HeartbeatFailure : The server detects a silent disconnection and forcibly removes the user to maintain game consistency.

LeaderElected : When the current session Leader disconnects, the system automatically assigns administrative privilege (e.g., the `/start` command) to the next available user.

Game Loop & Flow

GameStarted : The Leader initiates the session. A theme is loaded from the JSON source, roles are assigned (1 Narrator, N-1 Writers).

SegmentStarted : A new turn begins. The server starts the **Writing Timer** and broadcasts the input unlock signal to active Writers.

ProposalSubmitted : A Writer submits a sentence. The server send to the narrator all the proposal that were submitted in the writing phase.

ProposalSelected : The Narrator selects the winning proposal via its ID. The story is updated and broadcast to all connected clients.

Timeouts & Automation (Server-Authoritative)

WritingTimeout : The writing timer expires. The server forces the end of the submission phase, forwarding any existing proposals (or a system filler if none exist) to the Narrator.

NarratorTimeout (Auto-Resolution) : The Narrator fails to choose a proposal within the time limit. The server randomly selects a proposal and automatically advances to the next segment to prevent deadlocks (Auto-Continue).

Termination & Consensus

GameEnded : Triggered by the Narrator (via the continue or finish command). The system transitions to the Voting Phase for continue game in another round.

GameAborted : Triggered specifically when the Narrator disconnects. The current game state is wiped, and all users are returned to the Lobby immediately.

VotingSessionConcluded : Triggered when all users have cast their vote (Continue/Finish) or the voting timer has expired. If the consensus threshold is met, the system triggers a **LobbyReset** for a new game.

Message Types Exchanged

Communication follows a structured, JSON-based message protocol defined in the shared library. Messages are categorized into **Commands** (Client → Server) and **Events** (Server → Clients).

Commands (Client → Server)

- `JOIN_STORY(username)`: Request to enter the lobby or reconnect to an active game (State Recovery).
- `HEARTBEAT()`: Periodic signal to maintain the connection and prevent server-side timeout.
- `START_GAME()`: Sent by the **Leader** to initiate a new session.
- `SUBMIT_PROPOSAL(text)`: Sent by a Writer to propose content for the current segment.
- `SELECT_PROPOSAL(proposalID)`: Sent by the **Narrator** to choose the winning segment.
- `DECIDE_CONTINUE(action)`: Sent by the Narrator to choose whether to continue the story or end it (`CONTINUE` / `FINISH`).
- `VOTE_RESTART()`: Casts a "Yes" vote to return to the lobby after a game ends.
- `VOTE_NO()`: Casts a "No" vote, signaling the intent to disconnect.

Events (Server → Clients)

- `WELCOME(msg, is_leader)`: Confirms connection and assigns Leader status if applicable.
- `LEADER_UPDATE(msg)`: Notifies a client that they have been promoted to Leader.
- `GAME_STARTED(narrator, theme, role, is_spectator)`: Signals the start of a story and assigns roles. Also used for synchronization during reconnection.
- `NEW_SEGMENT(segment_id, timeout)`: Starts a new turn, unlocks input for writers, and synchronizes the client-side timer.
- `NARRATOR_DECISION_NEEDED(proposals, timeout)`: Sent to the Narrator when the writing phase ends.
- `STORY_UPDATE(story_list)`: Broadcasts the updated full text of the story to all clients.
- `ASK_CONTINUE(timeout)`: Asks the Narrator if the story should proceed.

- `GAME_ENDED(final_story, timeout)`: Signals the end of the narrative and opens the voting session.
- `VOTE_UPDATE(count, needed)`: Real-time update on the voting progress.
- `RETURN_TO_LOBBY(msg)`: Resets the client state to the lobby (triggered by vote success or critical failure).

3.4 Interaction

Communication occurs over persistent TCP connections using a custom JSON-based protocol with fixed-header framing. To ensure connection stability, clients and server exchange periodic **Heartbeat** signals. The interaction flow is event-driven and server-authoritative.

On Story Start

- The **Leader** sends the start command.
- The Server initializes the `GameState`, selects a random **Theme** from the data source, and assigns roles (1 Narrator, N-1 Writers) based on the connected users.
- The Server broadcasts `GAME_STARTED` to all clients, synchronizing their UI.

During a Segment (The Game Loop)

- **Start:** The Server broadcasts `NEW_SEGMENT` and starts the **Writing Timer** (e.g., 60s).
- **Submission:** Writers submit proposals.
- **Completion (Standard):** When all active writers have submitted, the Server immediately stops the timer and sends the aggregated list of proposals **only to the Narrator** via `NARRATOR_DECISION_NEEDED`.
- **Completion (Timeout):** If the timer expires, the Server forces the end of the phase and forwards existing proposals (or a system filler) to the Narrator.
- **Selection:** The Narrator sends `SELECT_PROPOSAL`. The Server broadcasts `STORY_UPDATE`.

Narrator Decision & Flow Control

- After an update, the Server asks the Narrator (`ASK_CONTINUE`) whether to proceed or end the story.
- If the Narrator fails to respond within the timeout, the Server executes an **Auto-Continue** strategy to prevent deadlocks.

On Story Completion (Consensus Phase)

- When the story ends, the Server triggers `GAME_ENDED` and a **Voting Session** begins.
- Clients send `VOTE_RESTART` or `VOTE_NO`. The Server monitors the vote count.
- If consensus is reached, the Server broadcasts `RETURN_TO_LOBBY`. Users who voted "No" are gracefully disconnected.

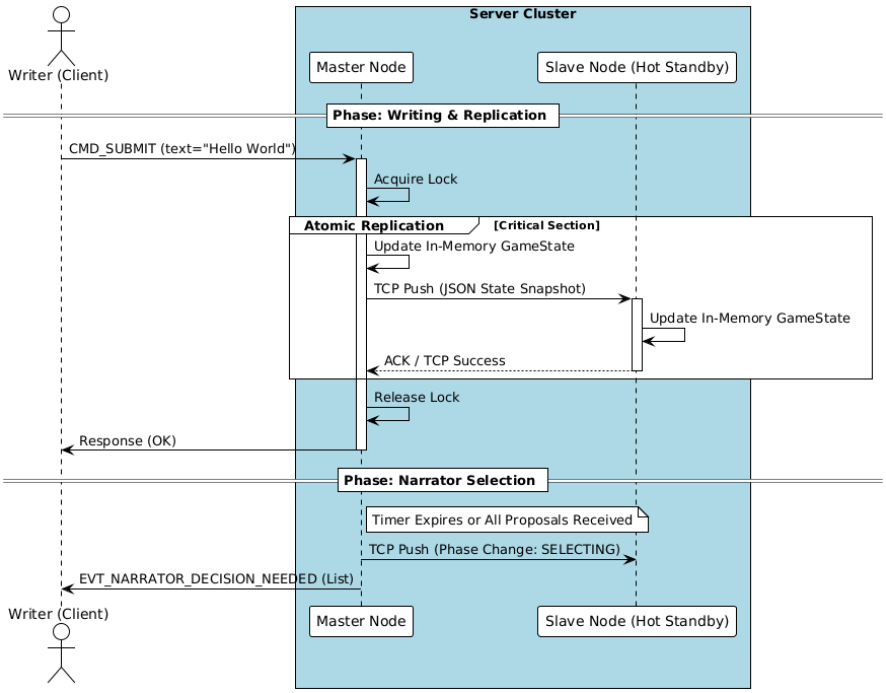

Figure 3: Sequence diagram Client → Server

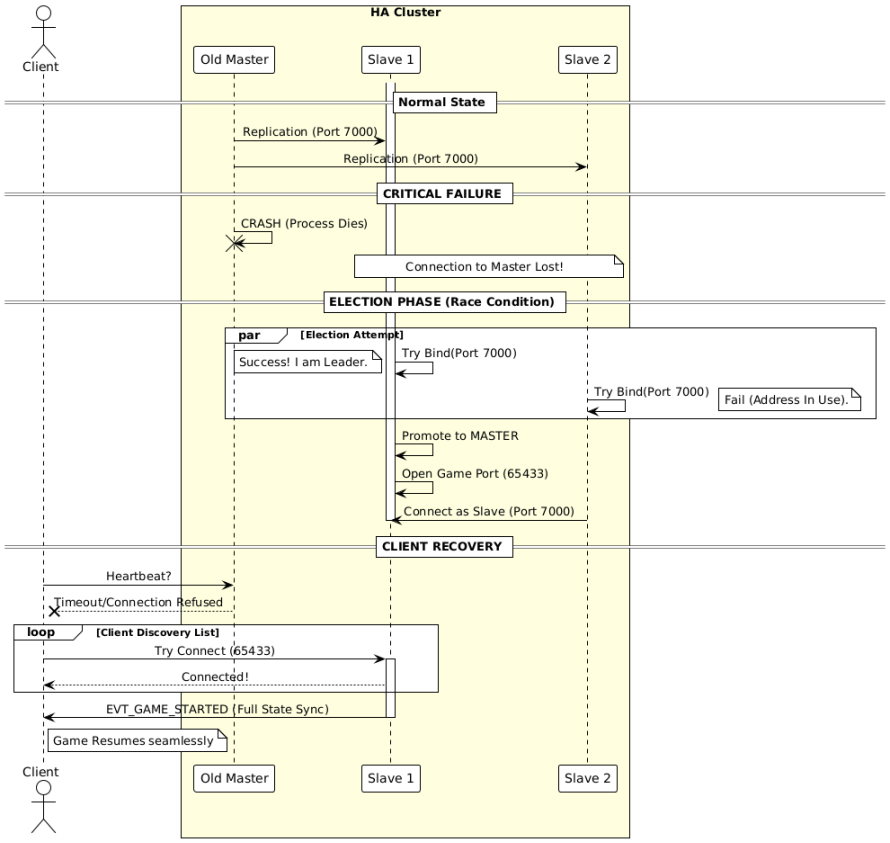

Figure 4: Sequence diagram Server

3.5 Behaviour

Components exhibit both stateless and stateful behaviours depending on their responsibilities.

Server Behaviour

The Server acts as the central **Controller** and **Model** manager (`GameState`). It is authoritative, meaning it holds the "single source of truth" and enforces rules, timings, and state transitions.

Stateful Responsibilities The server maintains the persistent global state, specifically:

- **Active Connections:** A map of sockets to usernames, continuously monitored for liveness.
- **Session Whitelist (`story_usernames`):** A snapshot of players allowed to interact (writers/narrator), essential for distinguishing active players from spectators and enabling state recovery.

- **Game Context:** The current story content, active theme, narrator identity, and current segment ID.
- **Transient State:** Pending proposals for the current round and active timers.

Connection & Lifecycle Management

- **Active Monitoring (Heartbeat):** A background thread checks the last activity timestamp of every client.
- **Smart Reconnection (`CMD_JOIN`):** When a user connects, the server checks the whitelist. If the user is recognized, the server sends the full game state and unlocks their input (Recovery). If new, the user is added as a Spectator.
- **Fault Tolerance Handling:**
  - *Writer Failure:* Round continue and writer's username is saved in the whitelist.
  - *Narrator Failure:* The server detects the critical failure, aborts the game instance, and forces a reset to the Lobby.
  - *Leader Failure:* The role is automatically reassigned to the next available user.
  - *Server Failure:* Watchdogs manage the failure of the server.

Game Loop Orchestration

- **Story Initialization:** Triggered by the Leader. The server resets the state, loads a Theme, assigns roles, and broadcasts the start signal.
- **Segment Flow & Timers:** The server manages the **Writing Timer**. If all active writers submit, the timer stops early. If the timer expires, the server forces the transition to the selection phase.
- **Narrator Interaction:** The server sends proposals only to the Narrator. It enforces a **Selection Timer**; if expired, the server performs an **Auto-Selection** (random) and an **Auto-Continue** to ensure liveness.

Termination & Consensus When the story ends, the server initiates a **Voting Phase**. It collects votes and, upon reaching consensus (or timeout), either resets the lobby for a new match or disconnects users based on their choice.

Persistence & Failure Model

- **In-Memory State Volatility:** The core game logic relies on in-memory structures. Consequently, a server crash results in the loss of the active session context (current segment, active timers).

- **Automatic Service Recovery (Watchdog):** To mitigate service unavailability, the server application is wrapped by a supervisor process (`runner.py`). This Watchdog monitors the process lifecycle: upon detecting a crash (non-zero exit code), it automatically restarts the server instance after a brief cooldown (3s), restoring the Lobby availability without human intervention.
- **Data Persistence:** To guarantee the durability of successful matches, completed stories are serialized to JSON files in the persistent storage immediately upon the `StoryCompleted` event.

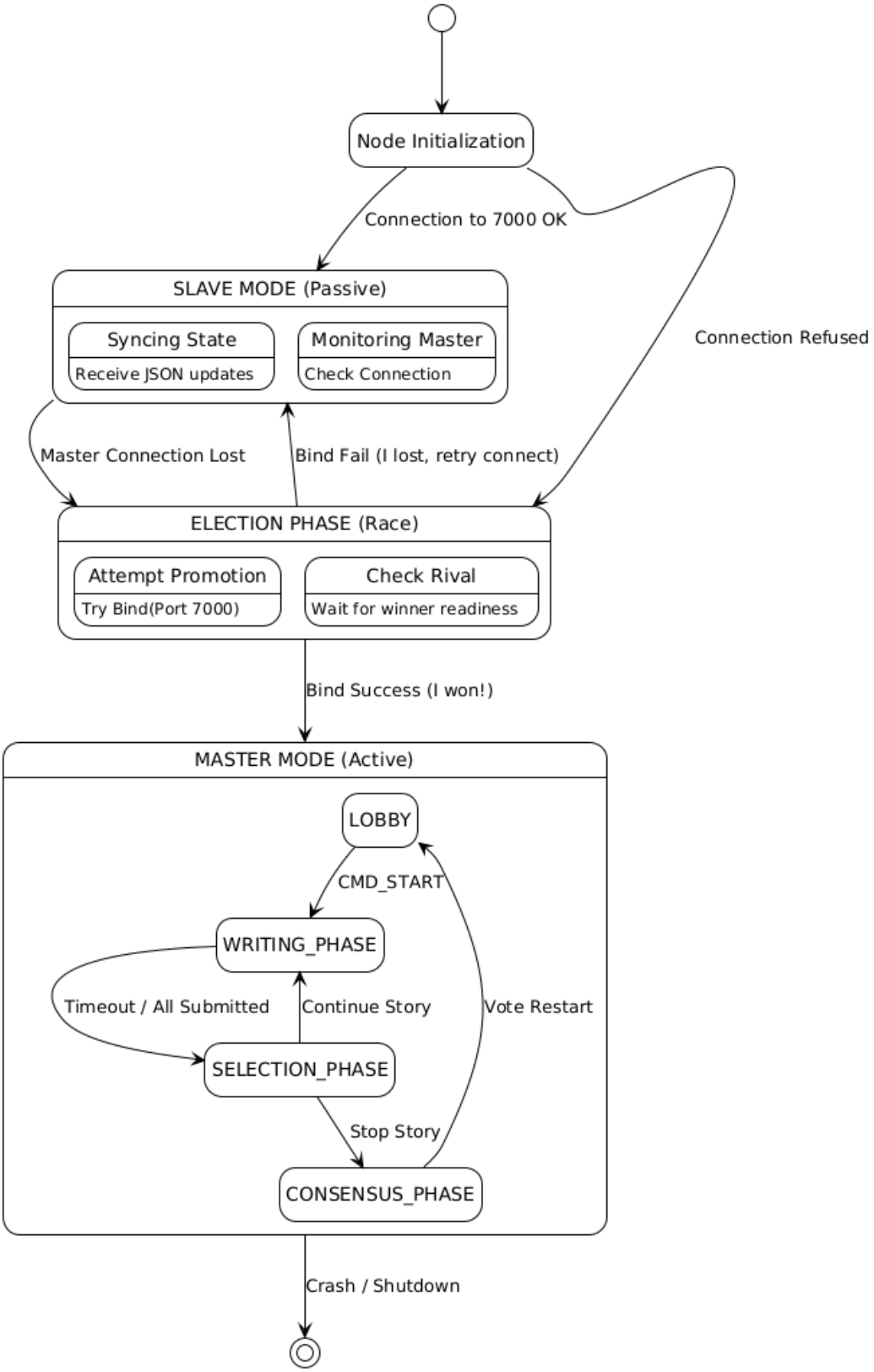

Figure 5: State diagram server

Player Behaviour

Each Player Client is a stateful distributed component that maintains a local view of the system state. The architecture is **Event-Driven**: the client does not execute game logic locally but updates its UI and input permissions solely in response to server events.

Local Client State The client maintains a lightweight state machine synchronized with the server:

- **Identity & Roles:** Tracks attributes such as `is_leader`, `am_i_narrator`, and `is_spectator`.
- **Game Phase:** Tracks the current interaction mode (e.g., `EDITING`, `DECIDING`, `VOTING`, `VIEWING`).
- **Visual Timer:** A local countdown synchronized with the server's timeout value to provide user feedback.
- **Story Replica:** A local copy of the approved narrative text for display.

Reaction to Server Events The client listens for JSON events and transitions between states:

`GAME_STARTED` Stores assigned roles, clears the interface, and displays the Theme.

`NEW_SEGMENT` Starts the local **Visual Timer**. If the user is a **Writer**, it transitions to `EDITING` (unlocking input); otherwise, it enforces `VIEWING` (read-only).

`NARRATOR_DECISION_NEEDED` Transitions the Narrator to the `DECIDING` state, displaying the list of proposals.

`STORY_UPDATE` Appends the new authorized text to the local story view.

`GAME_ENDED` Transitions all clients to the `VOTING` state, enabling consensus input.

`RETURN_TO_LOBBY` Resets local state flags. If the user is the Leader, the start command is enabled.

Commands Sent to Server

- **Game Actions:** Commands like `SUBMIT_PROPOSAL` or `SELECT_PROPOSAL` are sent based on user input. Sending an action immediately locks the local UI and stops the local timer to prevent duplicate submissions.
- **Background Signals:** A dedicated thread sends a `HEARTBEAT` signal to maintain the connection.

Connection & Recovery Strategy The client supports **State Recovery**. If a user restarts the application and rejoins with the same username, the client blindly accepts the state provided by the server (Role, History, Current Turn). This allows a Writer to immediately resume the `EDITING` phase without losing progress.

Narrator Behaviour

The Narrator is a temporary, privileged role assigned to one Player for the duration of the story. The Narrator client inherits all standard Player behaviours but gains exclusive states for **Conflict Resolution** and **Flow Control**.

Extended State When the `am_i_narrator` flag is active, the client maintains:

- **Candidate Pool:** A transient list of competing proposals received from the server at the end of a writing phase.
- **Flow Control Authority:** The ability to determine whether the narrative concludes or proceeds to the next segment via the `DECIDE_CONTINUE` command.

Reaction to Server Events

`NEW_SEGMENT` Unlike Writers, the Narrator enters a **Passive State** (`VIEWING`). The input is locked while waiting for writers to submit their content.

`NARRATOR_DECISION_NEEDED` Triggered when the writing phase is complete. The narrator transitions to the **DECIDING** state, displaying the list of proposals and enabling selection controls. A timer indicates the deadline before server-side auto-selection occurs.

`ASK_CONTINUE` Triggered after a selection. The client transitions to the **DECIDING_CONTINUE** state, prompting the user to choose between "Continue Story" or "End Game".

Commands Sent to Server

`SELECT_PROPOSAL(proposalID)` Transmits the ID of the chosen segment to the server and locks the UI.

`DECIDE_CONTINUE(action)` Sends `CONTINUE` to trigger a new segment generation, or `STOP` to trigger game termination and the voting phase.

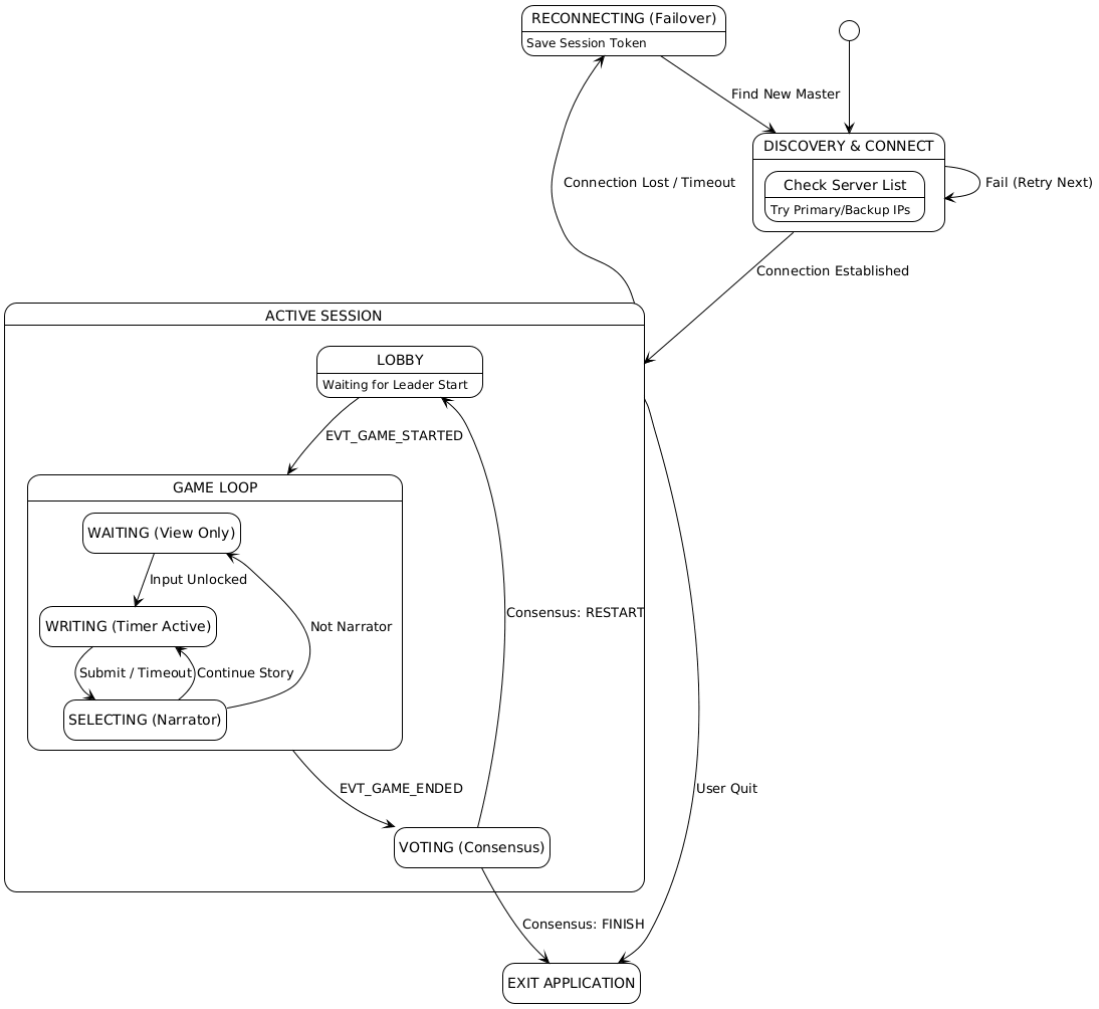

Figure 6: State diagram writer and narrator

3.6 Data and Consistency Issues

This section outlines how the system manages data integrity, redundancy, and synchronization challenges in the distributed Master-Slave environment.

Persistence & Redundancy The active game state resides in the server's volatile memory (RAM) to ensure low latency. The system implements **Active In-Memory Replication**: every state change on the Master is immediately serialized and pushed to all connected Slave nodes. This ensures that the active session data survives even if the Master process terminates unexpectedly. Long-term data durability for successful matches is further guaranteed by serializing completed stories to JSON files on the active Master's file system.

Consistency Issues Consistency is managed on two levels:

- **Client-Server (View Consistency):** The system enforces **Strong Consistency** via a Server-Authoritative model. Clients strictly render the state provided by the Master node and do not perform speculative local logic.
- **Intra-Cluster (State Consistency):** Synchronization between Master and Slaves occurs in real-time. The Master pushes "State Snapshots" via the replication channel immediately after processing a state-changing event, ensuring that Slaves are always aligned with the latest authoritative state.

Concurrency Issues Simultaneous interactions, such as multiple writers submitting proposals at the same instant, are managed through **Master-Side Locking**. A global reentrant lock (`threading.RLock`) serializes access to shared data structures. Crucially, this lock also covers the **Replication Step**: the state is synchronized to Slaves *before* the lock is released, ensuring that all nodes observe state transitions atomically.

3.7 Fault Tolerance

The system architecture has been upgraded to a **Multi-Layered Fault Tolerance** model. While maintaining liveness for logic errors, it now provides full redundancy against hardware or network failures, addressing the limitations of the previous single-server design.

Layer 1: Software Resilience (Watchdog Pattern) To handle transient software bugs (e.g., unhandled exceptions leading to a crash), the application is wrapped by a **Watchdog** supervisor process (`runner.py`).

- **Automatic Restart:** The Watchdog monitors the process lifecycle. Upon detecting a crash (non-zero exit code), it restarts the instance locally after a 3-second cooldown. This resolves "soft" failures without requiring a full cluster failover.

Layer 2: Infrastructure Resilience (HA Cluster) To handle critical node failures (e.g., machine shutdown, network partition) where a simple restart is insufficient, the system employs a **Master-Slave Failover** strategy:

- **Hot Standby Replication:** Slave nodes continuously receive state updates. If the Master fails, no data is lost as the state is already present in the Slaves' memory.
- **Atomic Leader Election (The "Port Race"):** Upon detecting the loss of the Master (Replication Channel closed), all Slave nodes initiate an election procedure. They attempt to bind the designated **Replication Port (7000)**. Relying on OS-level atomicity, only one node succeeds; the winner promotes itself to Master, while others reconnect to the new leader.

Client-Side Failover Clients are no longer bound to a single static endpoint. They utilize a **List-based Discovery** mechanism (e.g., `[Primary, Backup1, Backup2]`). If the connection to the current Master is lost, the Client automatically iterates through its configuration list until it establishes a connection with the new active Master, rendering the failover transparent to the user.

Connection Monitoring (Active Heartbeat) To detect "zombie" connections instantly:

- **Client-Side:** A background thread transmits periodic `CMD_HEARTBEAT` signals.
- **Server-Side:** A monitor thread checks user activity timestamps. If a timeout occurs, the resources are freed.

Logic Fault Tolerance (Deadlock Prevention) To ensure the game loop never stalls due to AFK players or network lag, the server employs **Authoritative Timers**:

- **Timeouts:** If a Writer or Narrator fails to act within the time limit, the server forces a state transition (e.g., Auto-Selection of a random proposal) to preserve game flow.

3.8 Availability

High Availability Architecture While the game logic remains centralized to ensure strict causal consistency, the physical infrastructure is replicated. No Load Balancer is required because the Client-Side Discovery mechanism handles the routing logic, automatically directing traffic to the currently active Master node from a prioritized list.

Service Continuity To guarantee continuous uptime:

- **Hot Standby Redundancy:** The system maintains N Slave nodes that hold a real-time copy of the game state.
- **Fast Failover:** In the event of a Master node crash (whether software or hardware), the election mechanism promotes a Slave within seconds (approx. 2-5s). This drastically reduces the Mean Time To Recovery (MTTR) compared to a cold restart.
- **Zero-Data-Loss Handover:** Since Slaves are synchronized synchronously before unlocking the next game phase, the promoted Master resumes the session exactly where the previous one left off, ensuring seamless continuity for players.

Network Partitioning Behaviour The system prioritizes **Liveness** in the event of network instabilities:

- **Client Partitioning:** The server detects unreachable clients via the **Heartbeat Monitor**. It automatically adjusts the **Dynamic Quorum**: if a writer drops due to a partition, the required number of proposals decreases, ensuring the game loop does not freeze for the remaining connected players.
- **Server Partitioning (Split-Brain Mitigation):** The atomic nature of the OS-level Port Binding (Port 7000) acts as a tie-breaker. It prevents multiple nodes from assuming the Master role simultaneously within the same network segment, ensuring that only one authoritative source of truth exists.
- **Recovery (Full Sync):** When connectivity is restored, clients reconnect using their username. The new Master responds with a **Full State Transfer**, synchronizing the client with the current story progress and restoring their role instantly.

4 Implementation

This section details the technological choices made to translate the distributed HA architecture into a working system. The implementation prioritizes **simplicity, portability, and minimal dependencies**, relying exclusively on the Python Standard Library.

Network Protocol: Dual-Channel TCP We selected **TCP (Transmission Control Protocol)**. The implementation manages two distinct communication channels:

- **Game Channel (Public):** Used for Client-Server interaction (e.g., Port 65432). It handles stateful connections, enabling immediate detection of disconnections via socket errors.
- **Replication Channel (Internal):** A dedicated TCP stream (Port 7000) used exclusively for Master-to-Slave synchronization. This channel requires low latency to minimize the window of potential data loss during a failover.

Data Representation & Framing In-transit data is serialized using **JSON** encoded in UTF-8. This choice ensures interoperability and human readability.
Stream Handling (Framing): Since TCP is stream-oriented, we implemented a **Length-Prefix Framing** protocol. This ensures the receiver (both Clients and Slaves) can correctly distinguish where one JSON message ends and the next begins.

Data Storage Strategy The system employs a hybrid storage strategy:

- **Real-Time Persistence:** The active game state is stored in the Master's RAM and immediately replicated to the RAM of all connected Slave nodes via network serialization.
- **Long-Term Persistence:** Completed stories are serialized as `.json` files in the `data/` directory of the Master node. This meets the requirement to preserve history without the overhead of a database server.

Authentication & Authorization

- **Authentication:** Identity is claim-based via a unique `username`. The Master enforces uniqueness and replicates the session whitelist to Slaves, ensuring that a user can seamlessly reconnect to a new Master after a failover.
- **Authorization:** The system implements in-memory **Role-Based Access Control (RBAC)**. The `GameState` logic validates every command against the user's role.

Concurrency Model (Thread-per-Client + Replication) The server architecture has evolved to handle multiple concurrent responsibilities:

- **Client Threads:** A dedicated thread handles I/O for each connected player.
- **Replication Thread:** A specialized background thread manages the connection with Slave nodes.
- **Synchronization:** A global `threading.RLock` protects the shared `GameState`. Crucially, the lock is held during the State Update + Replication Push cycle. This atomicity ensures that Slaves never receive a partial or inconsistent state update.

4.1 Technological details

The project is developed entirely in **Python 3.x** without external frameworks.

- **Networking:** `socket` library for managing both Game and Replication TCP channels.
- **Concurrency:** `threading` for parallel execution of client handlers and the heartbeat monitor.
- **Serialization:** `json` for encoding payload and `struct` for binary packet framing.
- **GUI:** `tkinter` was chosen for the client interface as it provides a cross-platform, event-driven UI natively within Python.
- **Process Supervision:** `subprocess` is used by the Watchdog (`runner.py`) to manage the server lifecycle and restart processes upon failure.

5 Validation

The validation of the system was conducted through a hybrid approach combining extensive Automated Unit Testing for logic verification and Manual Acceptance Testing for runtime behavior, fault tolerance, and GUI responsiveness.

5.1 Automatic Testing

To ensure code quality and architectural integrity, we implemented a suite of **12 automated test modules** using the standard Python `unittest` framework. The testing strategy relies on **Mock Objects** (to simulate network sockets without establishing real connections) and **Monkey Patching** (to isolate the file system during logic tests).

5.1.1 Testing Strategy

- **Unit Testing of Individual Components:** The core logic class, `GameState`, was tested in isolation. We verified internal methods such as leader election algorithms, turn rotation, and player management without involving the network layer.
- **Integration and Communication Testing:** Interaction between components was tested using mocked sockets. The `test_protocol.py` module verifies that the custom JSON-over-TCP protocol correctly serializes messages and handles the length-prefix framing, ensuring that the Client and Server share a consistent data format.
- **End-to-End Simulation:** We simulated entire game sessions programmatically (`test_full_simulation.py`). This involves instantiating a Server logic and multiple virtual Clients within the same test process to verify the correct flow of data from the "Start Game" command to the final "Story Archiving", mimicking a production environment in memory.

5.1.2 Test Specifications

The following table details the rationale and coverage of the primary test modules:

Test Module	Rationale & Methodology	Requirement Tested
`test_persistence.py`	**Rationale:** Verifies that game state is correctly serialized to disk. **Auto:** Setup/Teardown creates/deletes temp JSON files.	*Fault Tolerance* (Crash Recovery)
`test_concurrency.py`	**Rationale:** Checks for Race Conditions when multiple users write simultaneously. **Auto:** Spawns 100 concurrent threads invoking server methods.	*Performance & Stability*
`test_protocol.py`	**Rationale:** Ensures packets are not corrupted. **Auto:** Uses a Mock Socket buffer to simulate TCP stream fragmentation.	*Communication Model*
`test_spectator.py`	**Rationale:** Ensures late-joiners cannot influence the game. **Auto:** Asserts that write-permission is denied for non-whitelisted users.	*Access Control*
`test_cleanup.py`	**Rationale:** Ensures no data leaks between matches. **Auto:** Checks memory state after `abort_game()`.	*Reliability*

Table 1: Summary of Automated Tests

5.1.3 Execution

The tests are automated via a discovery script. To run the full suite:

```
python -m unittest discover tests -v
```

5.2 Acceptance Test

While unit tests validate the logic, manual acceptance testing was performed to verify the system's behavior in the real execution environment, specifically focusing on the **Watchdog process**, **GUI feedback**, and **Physical Disconnections**.

5.2.1 Manual Test Cases

Evaluating the non-automatable tests, we opted for manual tests regarding graphic aspects and user experience. Manual verification provided immediate visual feedback on the user experience.

1. **Scenario 1: Master Crash & Cluster Failover**
    - **Action:** While a game was in progress (with 1 Master and 2 Slaves active), we forced a critical failure on the Active Master node using the debug "CRASH NOW" command.
    - **Observation:**
        a) The **Slave Nodes** immediately detected the loss of the replication stream (Port 7000).
        b) The Slaves initiated the election procedure; one Slave successfully bound the replication port and promoted itself to **New Master** (e.g., on Port 65433).
        c) The **Clients**, upon detecting the disconnection, automatically iterated through their pre-configured server list and successfully re-established connectivity with the promoted node.
    - **Outcome: PASSED**. The game session resumed seamlessly on the backup node with full state preservation, confirming the correctness of the *Hot Standby Replication* mechanism.
2. **Scenario 2: Timer Enforcement (AFK Player)**
    - **Action:** A writer client intentionally did not submit text before the 60-second timer expired.
    - **Observation:** The server correctly identified the timeout, assigned a placeholder text, and advanced the global state to the Voting phase without blocking other players.
    - **Outcome: PASSED**.
3. **Scenario 3: Client Hot-Swap**
    - **Action:** We closed a Client window during a game and restarted it, logging in with the same username.
    - **Observation:** The server recognized the user via the session whitelist. The client skipped the Lobby and immediately loaded the current story and turn interface.
    - **Outcome: PASSED**.

6 Deployment

This section provides a step-by-step guide to deploying the **Distributed Storytelling** system from scratch on a new machine. Since the project relies on standard Python libraries, the deployment process is lightweight and does not require containerization engines like Docker.

6.1 Prerequisites

To run both the Server and Client components, the host machine must meet the following software requirements:

- **Operating System:** Cross-platform (Windows 10/11, macOS, or Linux).
- **Language Runtime:** Python **3.x** (Tested on 3.10+).
- **Libraries:** The project utilizes only Python's **Standard Library**. No external dependencies (e.g., `pip install`) are required. Key modules used include:
  - `socket, threading, json, time, os, sys, struct` (Backend & Networking)
  - `tkinter` (GUI Client - usually pre-installed with Python)
  - `unittest` (Testing framework)

6.2 Installation & Configuration

The system does not require setting environment variables or complex configuration files, as it uses relative paths for data persistence.

1. **Clone the Repository:** Extract the project source code into a directory of your choice. Ensure the directory structure is preserved as follows:

```
/ProjectRoot
  |-- /src
  |    |-- /server (contains main.py, gamestate.py, runner.py)
  |    |-- /client (contains main.py, ui.py)
  |    |-- /common (contains protocol.py)
  |-- /data (contains themes.json)
  |-- /tests (contains unit tests)
```

2. **Data Initialization:** Ensure the `data/` folder exists in the root directory and contains `themes.json`. If missing, the server will automatically create the folder and load a default theme fallback, but providing a custom JSON file allows for themed gameplay.

7 User Guide

This section describes how to launch, and use the **Distributed Storytelling** application. The system consists of a central Server (managed by a Watchdog for fault tolerance) and multiple graphical Clients.

7.1 Server Startup (HA Cluster Infrastructure)

To fully enable the High Availability features and Fault Tolerance, the system requires the startup of a **Server Cluster** composed of one Master node and at least one Slave node.

- **Step 1: Start the Master Node**
  Open a terminal in the project root directory and execute:

  ```
  python server/runner.py
  ```

  Outcome: The terminal will display `[SERVER] Master attivo su 127.0.0.1:65432`. This node is now the Active Leader accepting connections.

- **Step 2: Start Slave Node(s)**
  To provide redundancy, open a **new terminal window** (keep the Master running) and launch a Slave instance on a different port (e.g., 65433):

  ```
  python server/runner.py SLAVE 65433
  ```

  Outcome: The terminal will display `[SLAVE] Trovato Master! Entro in modalità passiva`. This node is now functioning as a Hot Standby, replicating data in real-time.
  Optional: You can repeat Step 2 with different ports (e.g., 65434) to increase the redundancy level.

7.2 Client Connection (Login)

Each player participates in the story using the Graphical User Interface (GUI) or the Command Line Interface (CLI).

- **Instructions:**
  1. Launch the client, it can be done with:

     ```
     python client/ui.py
     ```

     If we want to use the GUI or

```
python client/main.py
```

If we want to use the CLI

2. When the application opens, enter your **Username** in the dialog box.
3. Click OK.

- **Expected Outcome:** If the user has selected the GUI the main interface opens. The status bar at the bottom indicate: `[Name] | User | VIEWING`. The first player to connect is automatically appointed as the **Leader**.

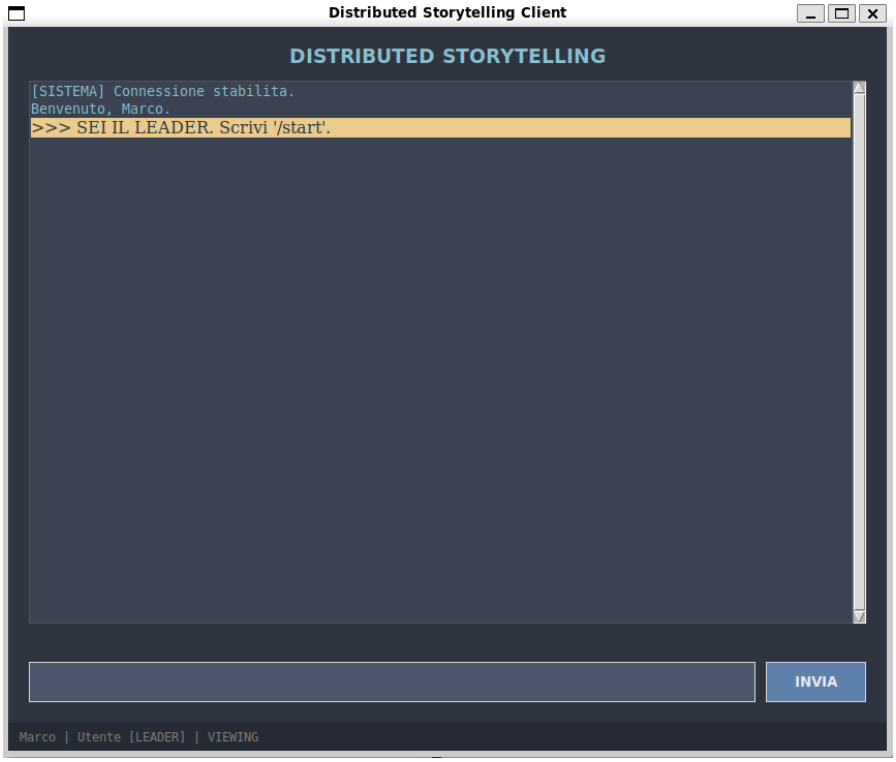

Figure 7: Login dialog and Client in Lobby state.

7.3 Starting the Game (Lobby Phase)

In this phase, players gather in the Lobby. Only the Leader has the permission to start the game.

- **Instructions (Leader):** Type the command `/start` in the input bar and press Enter or click "INVIA".
- **Expected Outcome:** The server randomly selects a **Theme** and assigns roles:

- One user becomes the **Narrator** (director of the turn).
- The others become **Writers**.
- Users joining after the start become **Spectators**.

7.4 Writing Phase (Writers)

During the game, the flow is regulated by a Server-Authoritative **Timer**.

- **Instructions:** Writers see a countdown (e.g., 60 seconds). They must write a sentence to continue the story and press Enter.
- **Expected Outcome:**
  - The input is sent to the server, and the text box is disabled.
  - The status changes to `WAITING`.
  - If the time expires without submission, the turn is forfeited.

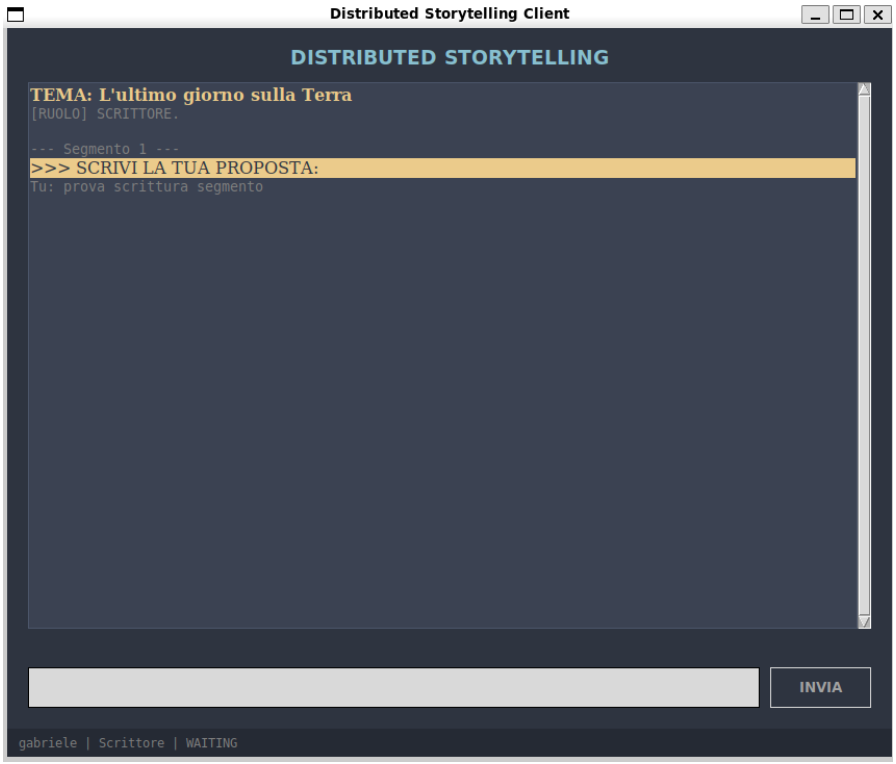

Figure 8: Writer interface with enabled input.

7.5 Selection Phase (Narrator)

The Narrator is responsible for choosing the best continuation.

- **Instructions:** The Narrator receives a list of anonymous proposals (e.g., `[0] Once upon a time...`, `[1] In a galaxy far away...`). The Narrator must type the **ID number** of the preferred proposal.
- **Expected Outcome:** The selected sentence is added to the global story. All clients receive the update (`STORY_UPDATE`), and a new turn begins.

Figure 9: Narrator interface showing the list of proposals.

7.6 Fault Tolerance

The system is designed to withstand unexpected disconnections.

Case A: Client Crash

- **Instructions:** If a client closes or loses connection, the user can simply reopen the application and enter the same username.
- **Outcome:** The server recognizes the user (via Whitelist) and immediately sends the entire story history and the current turn state, allowing the user to resume playing without interruption.

Case B: Server Crash

- **Scenario:** If the server process terminates unexpectedly.
- **Outcome:**
  1. The **Watchdog** automatically restarts the server within 3 seconds.
  2. The Server recovers the game state from the `recovery.json` file.
  3. Clients display the status `RECONNECTING...` (red bar) and automatically reconnect as soon as the server is online.

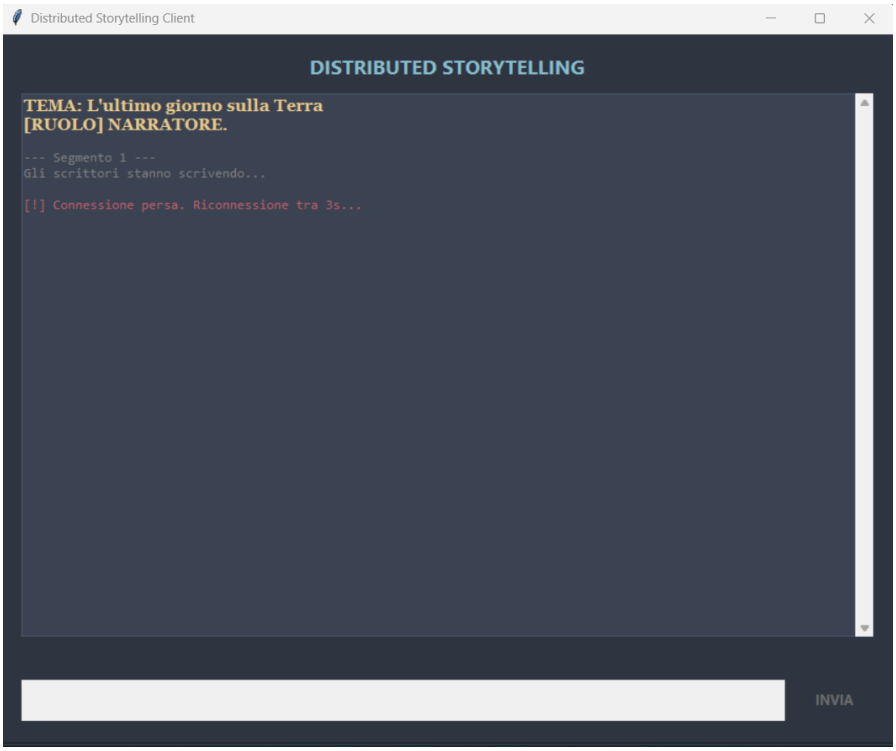

Figure 10: Client showing the red "RECONNECTING..." status bar.

7.7 Conclusion and Voting

The Narrator can decide to end the story.

- **Instructions:** When asked "Do you want to continue?", the Narrator types `F` (Finish).
- **Outcome:** The game ends. All players can vote:
  - `S` (Sì): Return to the Lobby for a new game.
  - `N` (No): Close the application.

8 Self-evaluation

8.1 Evaluation: [Marco Costantini]

Role within the group

In this project, my primary responsibility was the design and implementation of the **Backend Architecture** and the **Communication Protocol**. Specifically, I focused on:

- Designing the custom application-layer protocol (header + JSON payload) in `protocol.py` to ensure reliable message framing over TCP.
- Implementing the central `Server` logic and the `GameState` class, managing the synchronization of turn-based logic and state transitions.
- Developing the **Fault Tolerance** mechanism, including the JSON persistence layer (`recovery.json`) and the Watchdog script (`runner.py`) to handle automatic server restarts.

Product Evaluation

Strengths:

- **Robustness:** The system shows remarkable resilience. The combination of the watchdog and state recovery ensures that the service is highly available even in the event of critical crashes.
- **Modularity:** The separation between network logic (Sockets) and game logic (GameState) makes the code clean and maintainable.

Weaknesses:

- **Security:** Currently, all data is transmitted in plain text. Implementing SSL/TLS encryption would be necessary for a production environment.
- **Scalability:** While efficient for small groups, the use of a single-threaded event loop for game logic (protected by locks) might become a bottleneck with hundreds of concurrent users.

8.2 Evaluation: [Gabriele Arcese]

Role within the group

My role focused on the **Client-side Implementation**, **Concurrency Management**, and the **Validation Strategy**. My contributions included:

- Developing the Graphical User Interface (GUI) using `tkinter`, ensuring a responsive experience by decoupling network I/O from the UI rendering thread.

- Implementing the **Heartbeat** and **Timer** mechanisms, handling the complexity of multithreading to manage timeouts and keep-alive signals without freezing the application.
- Designing and writing the comprehensive **Automated Test Suite** (12 modules), covering unit testing, concurrency stress tests, and protocol fuzzing.

Product Evaluation

Strengths:

- **User Experience:** The client provides immediate feedback (timers, reconnection status bars) which significantly improves usability compared to a raw CLI.
- **Reliability:** The extensive test coverage provides a high degree of confidence in the system's stability, verifying edge cases that manual testing might miss.

Weaknesses:

- **Data Storage:** Relying on local JSON files for history storage is simple but lacks the query capabilities and performance of a real database (e.g., SQLite or PostgreSQL).
- **UI Aesthetics:** The interface, while functional, relies on standard widget libraries and lacks a modern, polished look.

9 Future Works

Although the current implementation fulfills all the primary requirements of a distributed system (transparency, openness, reliability), there are several areas where the project could be extended or optimized in future iterations.

9.1 Security Enhancements

Currently, the communication protocol transmits data in plain text (JSON over TCP). This makes the system vulnerable to packet sniffing and Man-in-the-Middle (MitM) attacks.

- **Proposed Implementation:** Wrap the existing TCP sockets with the Python `ssl` module to enforce TLS (Transport Layer Security) encryption. This would require generating self-signed certificates for the server and distributing the public key to clients.

9.2 Scalability and Database Integration

The current persistence layer relies on local JSON files (`recovery.json`, `history.json`). While efficient for small-scale sessions, this approach is not thread-safe for high-concurrency writes and lacks query capabilities.

- **Proposed Implementation:** Replace the file system storage with a lightweight relational database like **SQLite** or a distributed key-value store like **Redis**. This would allow for concurrent access management (ACID properties) and easier retrieval of historical data (e.g., "Find all stories narrated by Bob").

9.3 Unimplemented Feature: Peer-to-Peer Architecture

An interesting extension would be removing the central Server entirely, moving towards a decentralized P2P model.

- **Hypothetical Implementation:** Using a *Ring* or *Mesh* topology, the "Game State" could be replicated across all nodes using a consensus algorithm like **Raft** or **Paxos**. Leader election (for the Narrator role) would happen dynamically among peers, ensuring that the failure of any single node (even the current leader) does not stop the game.