Final Report Template
ACTS-PROJECT

Final Report for the Distributed Systems Course 2025/2026

cristian.morbidelli@studio.unibo.it
federico.morsucci@studio.unibo.it

July 20, 2026

ACTS is an advanced, agent-based smart city simulation focused on decentralized intersection management. Built in Python using the Mesa multi-agent framework and NetworkX, the project models dynamic urban traffic flow where intersections operate autonomously without a central command server.

At the core of the project is a procedural topology generator that constructs a directed graph representing a realistic road network, complete with external arterial roads and complex internal turning lanes. The simulation features two primary entities: Vehicle Agents that navigate the network, and Smart Traffic Light Agents deployed at intersection nodes.

Instead of relying on rigid, pre-programmed timers, the smart traffic lights utilize a distributed consensus algorithm to negotiate the "GREEN" state. Each agent continuously monitors its local environment, detecting queued vehicles, calculating waiting times, and receiving messages about incoming traffic waves from neighboring nodes. Based on this telemetry, each traffic light calculates a dynamic priority score.

To ensure physical safety and prevent collisions, the system implements Signal Phasing (conflict matrices). Traffic lights communicate their scores and intentions through a distributed messaging channel, utilizing Lamport clocks for event synchronization. Agents assigned to non-conflicting trajectories (identical phases) can concurrently grant themselves permission to turn green. Conflicting requests are dynamically resolved by comparing real-time traffic scores and enforcing minimum safety durations.

Ultimately, ACTS serves as a robust sandbox for testing distributed systems architectures, agent-based modeling, and modern traffic optimization algorithms in a highly reactive urban environment.

1 Concept

The project consists of a hybrid simulation environment for decentralized urban traffic management. Rather than a hardware-level distributed system, it is a **logically distributed simulation**: it models an IoT network in which each traffic light communicates as an independent node, but everything runs within a single environment to facilitate control and analysis.

The software components are divided into two main types:

- **Web Application (GUI Simulator):** It is the core of the project and is based on the Mesa framework. Mesa acts as a "time engine", advancing the agents step by step. The system includes a lightweight web server that displays the road map and traffic. Through HTTP APIs, it also provides a control panel allowing the user to manipulate the simulation live.
- **Command-Line Application (CLI):** It is a diagnostic tool (`monitor_traffic.py`) useful for observing what happens "behind the scenes". It connects directly to the message broker (Redis) to read messages in real time, allowing for the analysis and *debugging* of inter-agent communications without burdening the simulator's graphics.

Use Case Description

The system simulates the **peer-to-peer** coordination of an urban traffic light network. Even though the agents all run within a single process for simulation convenience, they behave exactly as they would in a real distributed system. To decide who has the right of way, they exchange asynchronous messages via an external broker (Redis) and use **Lamport clocks**[1] to guarantee the correct ordering of events. The effectiveness of these decisions is directly visible by observing the smooth movement of vehicles on the map; furthermore, the metrics and priorities of each individual traffic light can be read in real time by hovering over them with the mouse.

The architecture combines the Mesa simulator, a Redis instance isolated via **Docker**, and an interface accessible directly from the browser.

The interaction is designed to be highly flexible. Through the dashboard, the user can configure the initial environment, choosing whether to use a random map or load ready-to-use *demos*, created specifically to observe how the system resists bottlenecks or deadlocks. During the simulation, it is possible to modify the execution speed and the number of vehicles present on the fly. Beyond these global settings, the user can intervene on the fly to test the network's resilience, for example by forcing intersection shutdowns.

The system is intentionally **devoid of data persistence**: it does not use any traditional database. Information exists only temporarily in the simulator's memory; Redis acts exclusively as a "postman" (Pub/Sub Message Broker) to route JSON packets in real time.

From the basic panel, to verify the soundness of the algorithms, the user can choose from several preset simulations:

- **Procedural City:** A simulation that creates a different pseudo-random graph at each startup, where it is possible to adjust and select the number of cars.
- **Basic negotiation:** Shows how the basic agreement to cross intersections works.
- **Arrival order:** Demonstrates how the system tracks arrival times to establish priority for a given direction.
- **Fairness:** Verifies that waiting times are fair and that no vehicle remains blocked indefinitely, even in the presence of an infinite stream of high-priority cars.
- **Traffic wave:** Shows how neighboring intersections communicate to synchronize flows and create the green wave effect.
- **Lamport conflict:** Demonstrates how the system mathematically and lock-freely resolves requests that occur at the exact same moment.

Need for Distribution

Decentralized decision-making: Entrusting the traffic management of an entire city to a single central server would create a massive decision-making bottleneck[4]. By designing the system in a distributed manner instead, each intersection becomes autonomous and manages traffic by coordinating only with its immediate neighbors. The use of Mesa serves solely to provide a closed, predictable testing environment to measure the effectiveness of these distributed algorithms.

Fault Tolerance: A distributed architecture is vital to ensure stability by eliminating the risk of a single point of failure (*Single Point of Failure*). In our system, Redis acts as a common communication channel. If a traffic light fails, neighboring traffic lights detect it via timeouts (*timeouts* based on clocks or simulator *ticks*) and react autonomously by activating emergency mode (*failsafe*).

Real-world applicability: We chose distributed paradigms because traffic management is an inherently spatially distributed problem. This simulation serves as a testbed to demonstrate that event-driven techniques and mutual exclusion algorithms are the ideal solution for a real urban IoT network. The negotiation and communication code we developed could be transferred as-is to actual physical microcontrollers installed at every road intersection.

2 Requirements Elicitation and Analysis

This section defines the domain rules and system requirements, clearly explaining *what* the software must do without going into technical details of how it will be implemented.

2.1 Domain Glossary

To understand the system's logic, it is necessary to clarify the key concepts on which the simulation is based:

Agents and Steps: The environment is populated by autonomous entities (traffic light nodes and vehicles) called agents. Their actions and decisions are not continuous, but occur at discrete time intervals, called *steps* (or *ticks*).

Controlled Direction (Phase): A traffic light does not have a single color for the entire intersection, but manages several directions. A *Controlled Direction* is a group of lanes that can receive a green light simultaneously, ensuring that incompatible trajectories remain red.

Incoming Traffic Wave: This is the approaching traffic wave. Traffic lights communicate to their neighbors how many cars are about to arrive; this allows the system to predict traffic and assign priorities in advance, without waiting for cars to queue up.

Lamport Clock and Tie-breaking: Since traffic lights communicate asynchronously, they use *Lamport Clocks* to order events in time. These counters are used for *Tie-breaking*: if two traffic lights request a green light simultaneously and have the same traffic priority, the one with the lower clock wins.

Health Check and Failsafe Mode: To prevent a failure from blocking the entire city, nodes use a *Health Check* to verify that neighboring nodes are active. If a node stops responding for too long, adjacent intersections enter *Failsafe Mode* (e.g., flashing yellow), an emergency mode that allows traffic to flow anyway.

2.2 Functional Requirements

FR1 - Vehicle Navigation and Constraints: Vehicles must independently plan the route to their destination. Their movement is not instantaneous, but governed by physical road rules (travel time calculated as edge length divided by maximum speed). Furthermore, moving from one edge to another is strictly constrained by the traffic light state: a vehicle can cross the intersection only if the corresponding *Controlled Direction* is green.

FR2 - Decentralized Green Negotiation: Green allocation is not based on fixed timers, but on real traffic demand. Each traffic light calculates a score (*score*) by weighting the number of queued cars, accumulated waiting time, and incoming traffic waves communicated by neighbors. To switch its state to green, a traffic light must send a request and receive explicit authorization from all competing neighboring nodes.

FR3 - Starvation Prevention (Fairness): The system must guarantee fair treatment to all lanes, preventing a busy direction from monopolizing the intersection and blocking others. To this end, the traffic light is forced to yield the green once a maximum

time limit is exceeded, regardless of its *score*. Furthermore, after returning to red, it must observe a mandatory cooldown period before requesting passage again.

FR4 - Fault Detection and Failsafe Mode: The system must be resilient to network disruptions. Each traffic light monitors incoming messages from neighboring nodes. If a neighbor remains silent beyond a certain threshold, the traffic light triggers a *Health Check*. In case of no response, the intersection enters *Failsafe* mode (flashing yellow) to allow traffic flow regardless, interrupting normal negotiations until the failed node comes back online.

FR5 - Visualization and Operational Control: The software must provide a graphical interface for real-time observation of the road grid, vehicle positions, and traffic light colors. At the same time, it must offer operators the ability to test network robustness by simulating targeted failures: the user must be able to turn individual traffic lights off and on (via a *toggle power* command) and observe the resulting network adaptation.

2.3 Non-Functional Requirements

NFR1 - Deadlock Freedom and Resilience: In a distributed system, the blocking of a single component must never cause an irreversible freeze (*deadlock*) of the entire infrastructure. The system must guarantee that traffic continues to flow even in the case of severe anomalies (e.g., shutdown of multiple neighboring traffic lights), degrading performance but keeping the system alive and functioning through emergency modes.

NFR2 - Logical Consistency and Determinism: Since the road network is asynchronous and traffic lights do not share a global clock, the system must ensure that decisions are made in the same logical order for everyone. In case of simultaneous requests for the right of way, the system must resolve conflicts in a strictly deterministic manner, based on *Lamport Clocks* or, in case of a tie, on a predefined order (e.g., alphabetical ordering of IDs), preventing *race conditions*.

NFR3 - Simulation Fluidity: The simulation engine must be sufficiently performant to compute vehicle *pathfinding* and handle the intense exchange of Pub/Sub messages without causing freezes or prolonged graphical slowdowns. The system must maintain fluid *tick* progression to guarantee clear observation of the network's emergent behavior.

2.4 Implementation Requirements

IR1 - Multi-Agent Simulation Engine (Mesa): The entire domain logic (vehicles and traffic lights) must be written in Python and integrated within the Mesa framework. Time progression must be discrete and strictly governed by Mesa's *scheduler*. The use of threads or system timers to calculate movements or traffic light states is prohibited.

IR2 - Decoupling via Message Broker (Redis): It is strictly forbidden for an agent to directly invoke methods or read internal variables of another agent. To realistically simulate a real IoT network, every interaction between nodes must transit as

a message through an external Redis instance (via Pub/Sub pattern), running in an isolated Docker container.

IR3 - Inspectable Data Format (JSON): To enable real-time command-line monitoring and *debugging*, all payloads exchanged on the broker must be formatted in JSON. Binary formats or complex serializations are prohibited to keep network logs directly readable by a human operator.

3 Design

This chapter presents the architectural vision and design patterns employed to achieve the functional and non-functional requirements outlined in Section 2. System modeling was conducted following a ***technology-agnostic*** approach: structural decisions transcend specific implementation frameworks, taking root instead in the three fundamental pillars guiding the entire application domain: **control decentralization**, event-driven **asynchronous communication**, and **fault tolerance**.

3.1 Architecture

The **architectural style** of the system shown in Figure 1 combines two primary paradigms: **Multi-Agent Systems (MAS)**[5] and **Event-Driven Architecture (EDA)**[3].

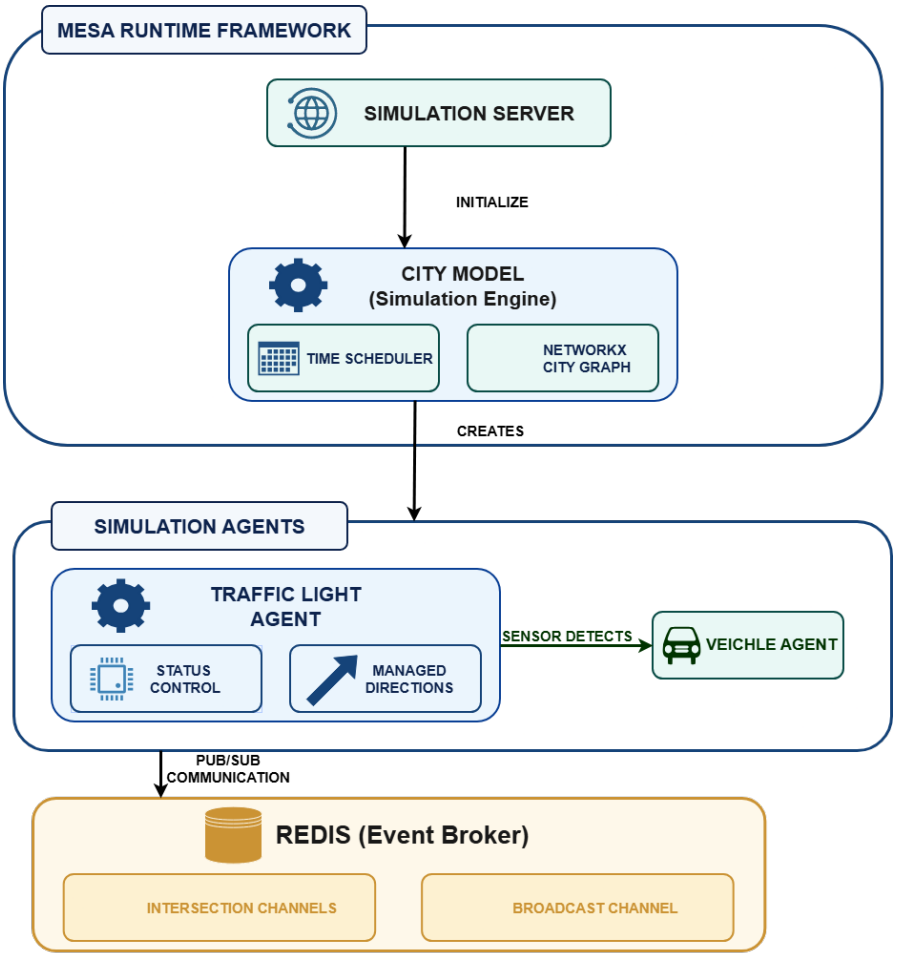

Figure 1: Architecture of the distributed system based on TrafficLight agents and a Redis broker.

We selected this approach to ensure that the system is reliable, scalable, and resilient to failures. Indeed, traditional centralized systems often risk creating bottlenecks and possess a single point of failure (*Single Point of Failure*, SPOF). Combining MAS and EDA overcomes these limitations, making the system significantly more reactive. To achieve this, we decomposed the application into numerous autonomous entities called agents. Each agent possesses an independent lifecycle, internal memory, and its own decision-making rules.

To ensure complete independence among agents, direct method calls were avoided, and no shared global memory was utilized. Agents communicate exclusively through **asynchronous message passing**. This logic adopts a **choreography**-based approach rather than orchestration. In other words, there is no supervisor node dictating and enforcing traffic light timings across the city; instead, each traffic light (`TrafficLightAgent`) makes autonomous decisions and holds equal authority. It manages only its own intersection and collaborates directly with neighboring traffic lights.

The use of the ***Publish/Subscribe*** pattern renders nodes decoupled in both space and time. For instance, when a vehicle or traffic light signals an impending queue by issuing an event (such as `TRAFFIC_SIGNAL`), it does not need to know who will receive the message or whether the recipients are online at that exact moment. Thanks to this fundamental decoupling, if a node experiences delays, fails, or temporarily disconnects, the rest of the network continues to operate normally, rendering the system highly robust.

Finally, the architecture maintains a **clear separation** between the physical space simulator and the data network. The simulator manages only the environment's "physics": spatial positions on the map, road travel times, and movement rules. The network, on the other hand, deals exclusively with routing information. Consequently, even though agents move within the spatial simulation, they reason and communicate using virtual network channels, behaving exactly as physical mini-computers deployed at real-world intersections would.

3.2 Infrastructure

The underlying infrastructure realizes the architecture described above, creating a modular environment where each component has well-defined responsibilities (*Separation of Concerns*). The system relies on four primary **infrastructural components**: the simulator, the *message broker*, the execution environment, and the user interface (client).

The **simulation layer** (which manages the spatial domain and system "physics") uses the *Mesa* library, a framework designed specifically for agent-based modeling (*Agent-Based Modeling*). This engine advances time and moves vehicles and traffic lights step by step. Alongside it, the road map is mathematically computed by *NetworkX*, which models it as a **directed graph** (*Directed Graph*). This map is essential both for vehicle pathfinding (*pathfinding*) and for enabling traffic lights to identify their topological neighbors.

The **network layer**, serving as the backbone for the event-driven architecture, is managed by the ***Redis*** message broker. Operating directly in RAM (*in-memory*), Redis creates an ultra-fast channel for messages (*Event Bus*). In this way, agents do not

communicate by passing variables within Python code, but instead exchange actual data packets over the network via TCP sockets.

This layer also resolves component naming and discovery (**Naming and Service Discovery**). Without resorting to DNS servers or central registries, the system uses a map-based (**topological**) addressing scheme. Each traffic light is assigned a unique identifier (ID) tied to its position (e.g., `tl_<node_id>`). To locate each other on the network, components automatically subscribe to their intersection's *Pub/Sub* channels (e.g., `channel_<intersection_id>`). In this way, local communications remain segregated from global alerts, which travel on a dedicated channel (`broadcast`). Structuring addresses into "channels" saves bandwidth and avoids sending unnecessary messages to all nodes, allowing nodes to communicate without needing prior knowledge of each other's IP addresses.

Regarding **network deployment** and the **execution environment**, a hybrid approach is adopted. Redis runs inside a container managed via *Docker Compose* for isolation and immediate readiness. Conversely, the simulator and agent code execute directly on the main host machine to maximize CPU utilization. Although everything runs on the same machine during simulation (communicating via *localhost*), the architecture is logically distributed and fully ***network-ready***. This means each agent could run on a separate hardware device (such as an *Edge/IoT* mini-computer at a real intersection) within a city network, without requiring any modifications to the project design.

Finally, the **presentation layer** acts as a *Client* and provides a graphical user interface via a Web page. Beyond displaying real-time traffic, this dashboard functions as a true ***Control Plane*** for system administrators: an operator can use it to communicate asynchronously with the network, such as injecting synthetic faults to test system response.

3.3 Modelling

To model the system, real-world road networks were abstracted into precise computer science concepts, defining domain entities, state representations, and message types.

The three primary **domain entities** of the system are:

- The `TrafficLightAgent` (the traffic signal), which performs computations and determines right-of-way priorities.
- The `VehicleAgent` (the vehicle), which navigates by computing optimal routes (*pathfinding*) and uses timers to manage its speed.
- The **Intersection** (along with its lanes), representing the shared resource subject to contention, mathematically modeled as a **directed graph**.

Regarding the **mapping to infrastructural components**, a dual nature exists. On one hand, the physical positions and movements of all agents are computed by the simulator (*Simulation Engine*) directly on the host machine. On the other hand, from the perspective of the distributed network, traffic lights communicate externally solely through the *Message Broker*. They operate as fully isolated processes: they never inspect

each other's memory, communicating strictly by sending and listening to messages on Redis channels.

The **state of the system** does not reside in a single centralized database; rather, it is partitioned temporarily across individual traffic lights. Within each `TrafficLightAgent`, the state consists of two parts:

- **Local State:** contains physical measurements, such as the number of queued vehicles (Q), total waiting time (W), and the current signal color or phase.
- **Distributed Coordination State:** provides the foundation for mutual agreement. It includes the ***Lamport Clock*** (a logical clock for establishing event ordering), the *permissions map* received from neighbors, and the ***Incoming Traffic Waves***, which represent predictive estimates of approaching vehicles.

Information modifying this state flows via structured **messages and domain events**, categorized into three types:

- **Negotiation Commands (*Commands*):** used to request intersection access. These include `REQUEST_GREEN` (by which a traffic light requests a green signal, transmitting its logical clock and urgency score) and `ALLOW_GREEN` (the explicit permission granted by neighbors if the maneuver is safe).
- **Domain Events (*Events*):** "fire-and-forget" broadcasts sent to update the network on current conditions. The most prominent is `TRAFFIC_SIGNAL`, through which a traffic light proactively notifies downstream neighbors about approaching vehicles, updating their *Incoming Waves*. These events are also consumed by the ***Traffic Monitor***, an external component that aggregates system messages for analysis and debugging.
- **Telemetry Queries (*Queries*):** underpin the system's fault-tolerance capabilities. They comprise the `HEALTH_CHECK` query (dispatched periodically to verify whether a silent node is operational) and its corresponding `ALIVE_SIGNAL` response. If a response fails to arrive, emergency routines isolate the faulty node.

3.4 Decision Strategy and Score Calculation

Traffic management relies directly on this state using a score calculated autonomously by each traffic light. The total score for a direction (S) considers not only currently queued vehicles but also **proactively accounts for** approaching traffic.

The formula used for this calculation is:

$$S = (Q \cdot 5 + W) + \left(\sum_{i=1}^{n} \frac{Wave_i \cdot \omega}{1 + \frac{ETA_i}{10}} \right) \cdot \frac{1}{|E|} \tag{1}$$

Where the main components are:

- **Local Component ($Q \cdot 5 + W$):** represents immediate urgency. Q is the number of queued vehicles and W is the total accumulated waiting time. Multiplying the queue length by 5 heavily prioritizes clearing congested lanes quickly.
- **Predictive Component (Incoming Waves):** adds the weight of incoming vehicle flows, made known via neighbors' `TRAFFIC_SIGNAL` messages. This value is weighted inversely by estimated arrival time (ETA_i), ensuring that physically closer vehicles have a stronger impact on the score.
- **Normalization Factor ($\frac{1}{|E|}$):** the weighting constant ω and division by the number of lanes ($|E|$) normalize calculations, preventing multi-lane approaches from unfairly dominating single-lane roads.

Through this formulation, traffic lights spontaneously form "green waves" (***coordinated green phases***) without requiring centralized command servers.

3.5 Interaction

Agent interaction, illustrated in Figure 3, serves as the primary engine of the system, transforming localized decisions into well-coordinated global behavior. Agents communicate **asynchronously** via events, eliminating blocking waits where nodes stall pending replies and avoiding the bottleneck risks inherent to synchronous architectures.

Managing the intersection as a shared resource requires preventing conflicting green phases (distributed mutual exclusion). To reach consensus, agents employ a **coordination protocol** free from central authority. When a traffic light requires a green phase, it transmits a `REQUEST_GREEN` message containing its urgency score and logical clock (***Lamport Clock***) to its topological neighbors. Neighbors grant permission (`ALLOW_GREEN`) only if switching to green poses no safety hazard or conflict with currently active signals, as depicted in Figure 2.

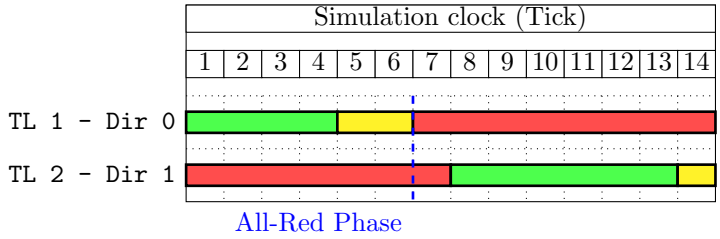

Figure 2: Temporal timeline showing mutex between two conflicting directions, including yellow light phase.

A fundamental aspect of this interaction is **deterministic tie-breaking**. If two traffic lights request a green light simultaneously with identical scores, a strict mathematical rule determines the winner: priority is granted to the lower Lamport Clock. If the logical clocks are also identical, priority falls to the agent with the smaller alphanumeric ID.

This logic guarantees deterministic agreement, eradicating system deadlocks or infinite yielding scenarios (*livelock*).

Beyond negotiating access permissions, the system actively disseminates traffic updates (**proactive information propagation**). Via `TRAFFIC_SIGNAL` messages, traffic lights notify downstream nodes in advance regarding vehicle counts and expected arrival times. This triggers a cascade effect allowing downstream intersections to prepare before vehicles reach the stop line, fostering emergent green waves.

Finally, communications are safeguarded by a **continuous resilience** mechanism based on a heartbeat pattern. Every agent monitors its neighbors' inactivity: if no messages are received within a defined threshold, it issues a `HEALTH_CHECK`. Absent a response, the system diagnoses a node failure and self-organizes: it isolates the compromised node and triggers safety procedures to ensure network continuity.

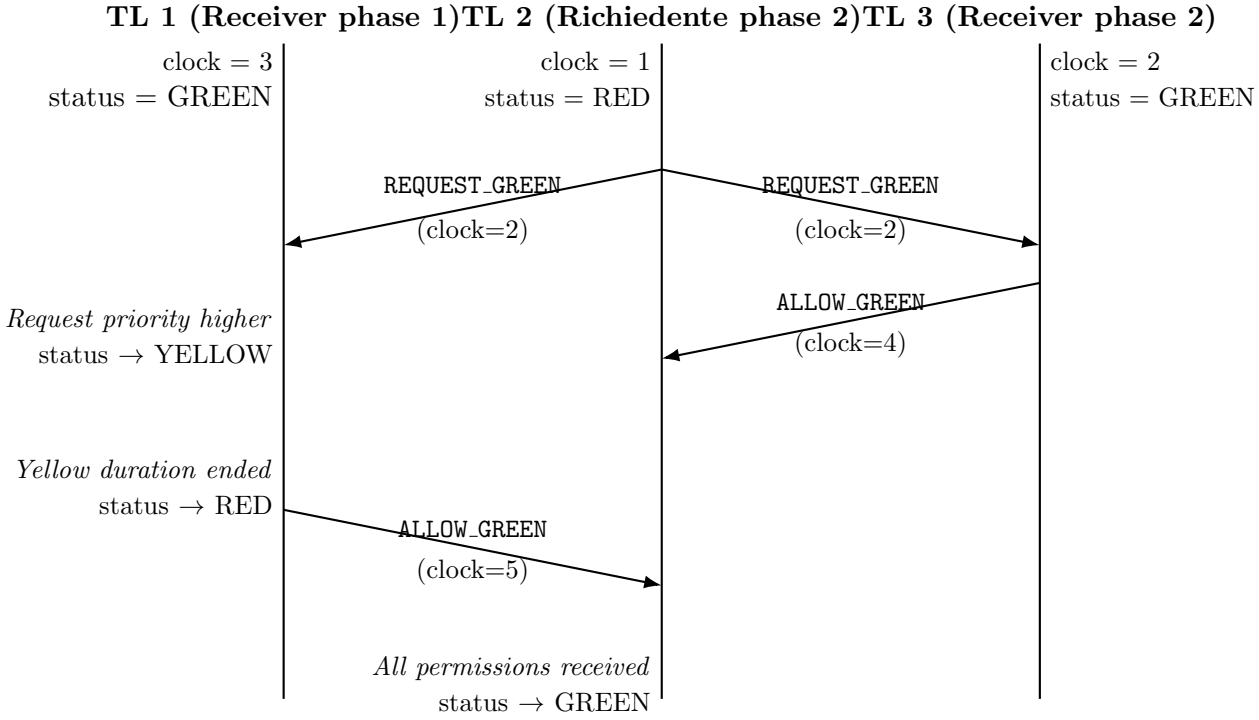

Figure 3: Sequence diagram with 3 traffic lights(TL 1, TL 2, TL 3).

3.6 Behaviour

The global behavior of the system arises from the continuous interaction between moving vehicles and traffic lights adapting to traffic. Each vehicle (`VehicleAgent`) operates using a **finite state machine** with two main phases: **QUEUED** (stopped in a queue, where it makes decisions) and **DRIVING** (in physical motion). When queued, the vehicle

computes its route using a search algorithm (*pathfinding*) that respects traffic rules. Departure, however, is not automatic: the system uses a **spatial locking mechanism** (a concurrency rule that allows only one vehicle to depart per node at a time) and requires an actual green signal toward the selected road. Once departed, the time and distance traveled are calculated mathematically based on speed limits and road length.

Regarding intersection control, the traffic light (`TrafficLightAgent`) does not rely on classic fixed timers, but instead leverages a **hybrid perception**. At each simulation step, the traffic light combines information it directly observes (by looking at nearby approaching cars) with data received from the network (the traffic "waves" signaled in advance by neighboring traffic lights). This allows it to prepare ahead of time and act proactively, rather than merely reacting at the last moment.

Traffic light color changes follow precise rules described by a finite state automaton shown in Figure 4, designed to guarantee the two fundamental properties of any distributed system: ***Safety*** and ***Liveness***.

- **Safety:** To ensure that accidents or conflicts never occur at the intersection, when the traffic light revokes the green signal, it must pass through a **transitional yellow**, followed by a pause period (*cooldown*) with a solid red signal. This provides mathematical certainty that the intersection has physically cleared before granting right of way to another direction.

- **Liveness:** To prevent a vehicle from being trapped waiting indefinitely (*starvation*), each time the light turns green, a maximum timer (`MAX_GREEN_TIME`) is started. If this time expires, the traffic light is forced to yield to other roads, even if it still holds a high score due to queue length. This guarantees everyone a maximum waiting time (*bounded wait*). Finally, a minimum green time also exists to prevent *thrashing*—excessively fast, wasteful color changes that would end up gridlocking traffic.

Finally, the system includes a feature to manage exceptional cases. Using the Web panel (*Control Plane*), a human operator can force a traffic light shutdown. When this occurs, the agent pauses its normal operation, safely freezes on solid red, and stops participating in shared decisions with other traffic lights (exiting the distributed consensus protocol). This action triggers an immediate reaction in moving vehicles as they realize that the intersection is blocked. All of this practically demonstrates the natural **resilience of decentralized behavior**, namely the system's ability to self-reorganize in real time when facing unexpected or drastic changes in the city map.

3.7 Data and Consistency Issues

Consistent with the ephemeral nature of a real-time simulation, the system does not require persistent data storage (no use of SQL/NoSQL databases). In-transit data management is delegated entirely to the *in-memory Message Broker*.

The real *consistency* challenges pertain to the **ordering of distributed events** (requirement NFR2).

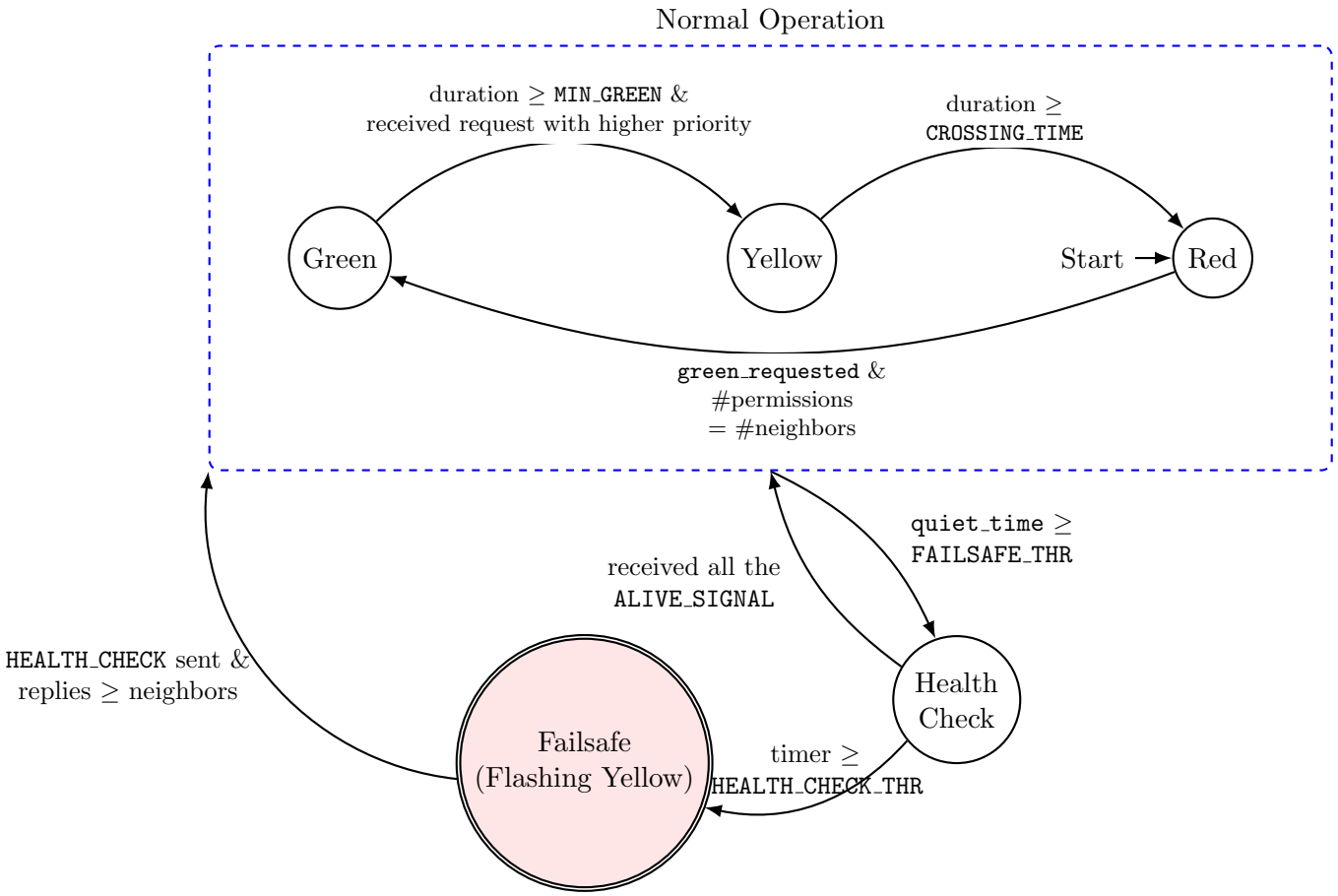

Figure 4: State transition graph of a traffic light with emergency protocol management

3.8 Fault-Tolerance

To satisfy the reliability requirement (NFR1) in an environment lacking a central controller, anomaly detection relies on a locally managed *timeout* pattern.

Each traffic light tracks the inactivity duration (*quiet time*) of its neighbors. If the critical threshold (`FAILSAFE_THRESHOLD`) is exceeded, the agent dispatches a `HEALTH_CHECK`. In the absence of a response (`ALIVE_SIGNAL`), the system detects a failure. Upon detecting such a failure, the agent isolates the compromised node and triggers a ***graceful degradation*** procedure, transitioning to an **emergency operating mode** (the ***Failsafe*** state, typically implemented via a **flashing yellow**) to keep vehicular traffic flowing—albeit degraded—and prevent permanent gridlocks.

3.9 Availability and Security

Operating within a *trusted* ecosystem (internal Docker networks), availability and security play a marginal role. No broker replication mechanisms or *load balancing* are implemented. System behavior in the event of a *network partitioning* aligns with fault-tolerance logic: nodes isolated by the partition independently enter the emergency state described above.

4 Implementation

Questo capitolo dettaglia le scelte tecnologiche e implementative adottate per la realizzazione pratica dell'architettura descritta nel Design. L'implementazione traduce i concetti di agenti, messaggi e clock logici in stack software operativi e protocolli di rete standardizzati.

Per quanto concerne i protocolli di comunicazione di rete, il sistema si affida a due standard differenti in base al contesto operativo. La comunicazione interna tra gli agenti e il message broker (Redis) avviene su protocollo **TCP/IP**, garantendo un trasporto affidabile e ordinato per i flussi di pubblicazione e sottoscrizione (pattern Pub/Sub). Parallelamente, l'interazione tra l'operatore umano e il pannello di controllo della simulazione sfrutta il protocollo **HTTP**. Nello specifico, il server espone endpoint RESTful (sulla rotta `/traffic-lights`) che accettano richieste `GET` per la lettura dello stato e `POST` per l'invio di comandi operativi, come l'induzione manuale di un guasto.

Tutti i dati in transito (*in-transit data*), sia sul broker di rete che nelle richieste HTTP, sono serializzati in formato **JSON** (*JavaScript Object Notation*). Questa scelta è motivata da due fattori principali: la sua natura *human-readable*, che si è rivelata fondamentale per l'ispezione dei pacchetti a riga di comando durante la fase di *debugging* della CLI, e la perfetta interoperabilità con i dizionari nativi del linguaggio Python, evitando il sovraccarico computazionale di librerie di serializzazione più complesse come Protocol Buffers. Ogni payload JSON trasporta i metadati essenziali della comunicazione distribuita, tra cui l'ID del mittente, il clock di Lamport corrente e il corpo del messaggio.

In linea con i requisiti del dominio, l'infrastruttura è priva di archiviazione persistente. Di conseguenza, non viene effettuata alcuna interrogazione a database relazionali (SQL) o documentali (NoSQL). L'unica base dati utilizzata è **Redis**, configurato però esclusivamente come *Message Broker in-memory*. Redis non memorizza lo storico dei messaggi, bensì si limita a instradarli dinamicamente in base alle iscrizioni attive sui vari canali (`traffic_channel`, canali di *broadcast* e canali locali degli incroci).

Per quanto riguarda i meccanismi di autenticazione (*authentication*) e autorizzazione (*authorization*), come OAuth, JWT o logiche RBAC, essi non sono stati implementati. Trattandosi di un Proof of Concept accademico eseguito in un ambiente di simulazione locale (container Docker interconnessi), la rete è considerata nativamente *trusted*. L'implementazione di schemi crittografici e di controllo degli accessi esula dagli scopi di questo specifico test di algoritmi di sincronizzazione; tuttavia, tali integrazioni diventerebbero un prerequisito fondamentale qualora il sistema venisse effettivamente tradotto in un ecosistema IoT pubblico su hardware fisico (aspetto che verrà ripreso nella sezione dei *Future Works*).

4.1 Technological details

L'intero nucleo logico del progetto è stato sviluppato in **Python 3.10**, sfruttando le potenti funzionalità di orientamento agli oggetti e la tipizzazione opzionale (*type hinting*) per modellare le strutture dati come `Request` e le macchine a stati.

Il cuore pulsante della simulazione è basato su **Mesa**[2], un framework open-source

specifico per la modellazione *Agent-Based* in Python. Mesa è stato sfruttato per due componenti critiche:

- **Lo Scheduler Temporale:** Che orchestra l'avanzamento discreto degli *step* temporali, richiamando ciclicamente la funzione `step()` di ogni semaforo e veicolo registrato nella simulazione.
- **Il Server Web Integrato:** Mesa incorpora internamente un server basato su **Tornado**, che è stato esteso in questo progetto per servire non solo il *canvas* grafico del grafo stradale, ma anche i summenzionati *handler* HTTP per il pannello di controllo manuale.

Il dispiegamento dell'infrastruttura (*deployment*) avviene tramite **Docker** e **Docker Compose**. Questa scelta tecnologica garantisce l'isolamento ambientale e la riproducibilità. Il file `docker-compose.yml` orchestra l'accensione simultanea di due servizi: il container ufficiale di Redis (esposto sulla porta `6379`) e il container applicativo della simulazione (esposto sulla porta `8521`), iniettando in quest'ultimo le variabili d'ambiente (es. `REDIS_HOST`) necessarie per la *Service Discovery* all'interno della rete virtuale locale.

5 Validation

The validation phase was designed to verify strict compliance with the outlined requirements, evaluating both the correctness of the domain logic and the robustness of the structural architecture. The process utilizes a combination of automated tests based on the *Pytest* framework, dedicated to algorithmic verification, and manual acceptance tests focused on evaluating the visual and interactive response of the system.

5.1 Automatic Testing

The automated testing strategy adopts a *white-box* approach. To ensure precise testing of internal logic, individual agents were isolated from the network infrastructure (Redis) and the spatial simulation engine (Mesa) by extensively employing the `unittest.mock` library, particularly through the `MagicMock` and `@patch` constructs. This isolation allows the injection of highly controlled scenarios. The entire suite is executed automatically upon system startup via the `pytest` command, regularly integrated into the `start.sh` script. The test architecture was divided thematically to cover the different functional domains of the system.

Pathfinding and Physical Constraints Validation (FR1)

The `test_vehicle.py` suite analyzes the lifecycle of the `VehicleAgent` in detail. The tests first confirm proper compliance with signals: given a red traffic light state (`tl_state = RED`) on the target edge, the vehicle maintains the logical condition of `QUEUED`. Upon the arrival of the green light, the agent consistently transitions to the `DRIVING` state, and the system calculates the travel time (`travel_timer`) strictly based on the physical parameters of the road. As an example, a scenario with a 30-meter long edge and a maximum allowed speed of 5 meters per *tick* yields an exact transit time of 6 *ticks*. At the end of the journey, the correct spatial reallocation of the vehicle onto the logical Mesa node and its return to the queued state are verified. Finally, to test system robustness under abnormal conditions, the `test_vehicle_handles_unreachable_destination` scenario evaluates the management of broken topologies or unreachable destinations: the system demonstrates its ability to formally intercept the `NetworkXNoPath` exception without crashing, forcing instead a safe and dynamic route recalculation.

Decentralized Negotiation and Phase Management (FR2, FR3)

The computational core of the infrastructure is validated within the `test_traffic_light.py` module. Here, the predictive score calculation is mathematically verified, ensuring that an intersection's switching priority adapts in real time by evaluating both physically waiting vehicles and the weight of *Incoming Traffic Waves* (approaching vehicles communicated by adjacent nodes). The distributed consensus mechanism is tested to guarantee the total absence of crossing collisions: switching to green light is inhibited unless the agent receives the `ALLOW_GREEN` event from all involved neighbors. At the same time, the suite demonstrates the effectiveness of anti-starvation mechanisms, confirming that a vehicular queue characterized by prolonged waiting times always receives priority over numerically superior but more recently formed flows, thus preserving the logical fairness of the entire network.

Logical Synchronization and Fault Tolerance (FR4, NFR1, NFR2)

In the absence of a global system clock, the tests specifically focus on validating causal ordering and fault tolerance. The robustness of Lamport Clocks for handling *tie-breaking* is demonstrated: any simultaneous mirror requests are deterministically resolved in favor of the lower logical clock or, secondarily, based on the alphanumeric value of the node identifier. Any permission or obsolete message received late is strictly discarded. State machine resilience (NFR1) is stress-tested by simulating an abnormal network silence from an adjacent node. This procedure confirms the correct transition through all emergency phases: exceeding the critical timeout, initiating the *Health Check* routine, and, if the silence persists, the agent's autonomous transition into *Failsafe Mode* (flashing yellow). Furthermore, the safe shutdown procedure (FR4) is validated: invoking `toggle_power` instantly cancels the node's pending negotiations, consistently isolating it from the rest of the network.

End-to-End Integration

Beyond in-memory algorithmic testing, the automated phase concludes with tests operating in the containerized environment managed by `docker-compose.yml`. By jointly starting the simulator and the Redis container, messages leave the test environment mocks to transit over real TCP sockets. This test architecture guarantees that the Pub/Sub communication flow operates correctly across the entire topology in a network ecosystem comparable to production deployment.

5.2 Acceptance Test

Although automated tests ensure high structural and logical code coverage, traffic flow dynamics and the clarity of user interaction require a qualitative empirical evaluation. To this end, algorithmic validations were complemented by targeted manual acceptance testing sessions.

The first phase of these tests is based on the visual execution of preset scenarios (such as the *Basic negotiation* and *Fairness* layouts). Prolonged observation of the simulations made it possible to empirically confirm that the local coordination rules implemented by individual agents successfully converge into efficient collective behaviors. The system demonstrates the ability to generate fluid "green waves", actively decongesting traffic jams anticipated in the case studies.

In parallel, the interaction experience between the human operator and the monitoring dashboard (FR5) was validated. By simulating direct network management via the web control panel, the system's response to induced fault injection was verified (e.g., the *Toggle Power* command for remote intersection deactivation). While unit tests certify correct code-level behavior, the acceptance test confirms that the interface properly receives the HTTP command and instantly updates the on-screen topological map. The interface promptly highlights the powered-off node and reports without delay the resulting transition of neighboring traffic lights into alarm state (Failsafe), demonstrating full synchronization between the backend distributed engine and the operational frontend.

6 Deployment

The distributed nature of the system and the need to separate the simulation engine from the *message broker* require a consistent execution environment. To simplify commissioning operations and ensure reproducibility across different machines, the system was designed to be launched via an automated process that integrates containerization and Python environment configuration.

6.1 Software Prerequisites

To run the system correctly, the following software components are required:

- **Docker Engine:** Version 20.10 or higher (required for the Redis instance).
- **Docker Compose:** For the orchestration of the message broker.
- **Python 3.10:** For running the local simulator and test suite.
- **Bash:** For executing the automated startup script.

6.2 Setup and Execution Procedure

The project integrates an automation script (`start.sh`) that manages the entire deployment lifecycle, from infrastructure configuration to code validation. The user can start the system by following these steps:

1. **Repository access:** Navigate to the root directory of the project.
2. **Infrastructure startup:** Run the startup script via the terminal:

```
./start.sh
```

Expected outcome: The script sequentially executes the following operations:

- Starts the Redis container in the background (via `docker compose up -d redis`).
- Creates a Python virtual environment (*venv*) and automatically installs the dependencies listed in `requirements.txt`.
- Runs the test suite with `pytest` to validate system integrity.
- Launches the simulator (`acts` module), making the graphical interface accessible via browser at `http://localhost:8521`.

6.3 Deployment Engineering

The approach to deployment reflects the need to maintain a rigorous and deterministic simulation environment.

Broker orchestration: Docker is used exclusively for the Redis instance. This choice allows isolating the communication infrastructure from the simulation process, ensuring that software agents interact solely through TCP network sockets, faithfully mimicking the decoupling of a real distributed system. The use of `docker compose` simplifies setup, eliminating the need to manually install and configure a Redis server on the host machine.

Python stack automation: The `start.sh` script implements an *environment hardening* logic. It automatically configures the `PYTHONPATH` to include the `src/` directory, transparently manages dependency installation, and ensures that the system is tested prior to each startup (*fail-fast approach*). In the event of missing dependencies or a test suite failure, the script aborts execution, guaranteeing that the simulator is launched only in a validated and consistent system state. This automation drastically reduces the risk of human error linked to incorrect local configurations, promoting project portability across different development machines.

7 User Guide

This section illustrates how to start, configure, and interact with the ACTS simulation environment. The system offers a web-accessible graphical user interface (GUI) for visual observation and a control panel for fault injection, complemented by command-line tools for network diagnostics.

7.1 Prerequisites and System Startup

The operation of the simulator requires the communication infrastructure to be active before launching the agents.

1. **Message Broker Startup:** Start the Redis container via Docker. This emulates the external network infrastructure:

   ```
   docker compose up -d
   ```

2. **Simulator Startup:** From the project root, launch the Mesa server:

   ```
   python -m acts
   ```

3. **Interface Access:** Open the browser and navigate to `http://localhost:8521`.

7.2 Main Interface and Configuration

Upon first access, the user is greeted by the main simulator interface (Figure 5). From the left side panel, environmental parameters can be dynamically configured before pressing the **Start** button.

- **Frames Per Second (FPS):** Adjusts the visual playback speed of the simulation.
- **Number of Cars:** Allows increasing or decreasing the traffic density introduced into the network.
- **Scenario Selection:** A drop-down menu allows selecting different preset simulation types.

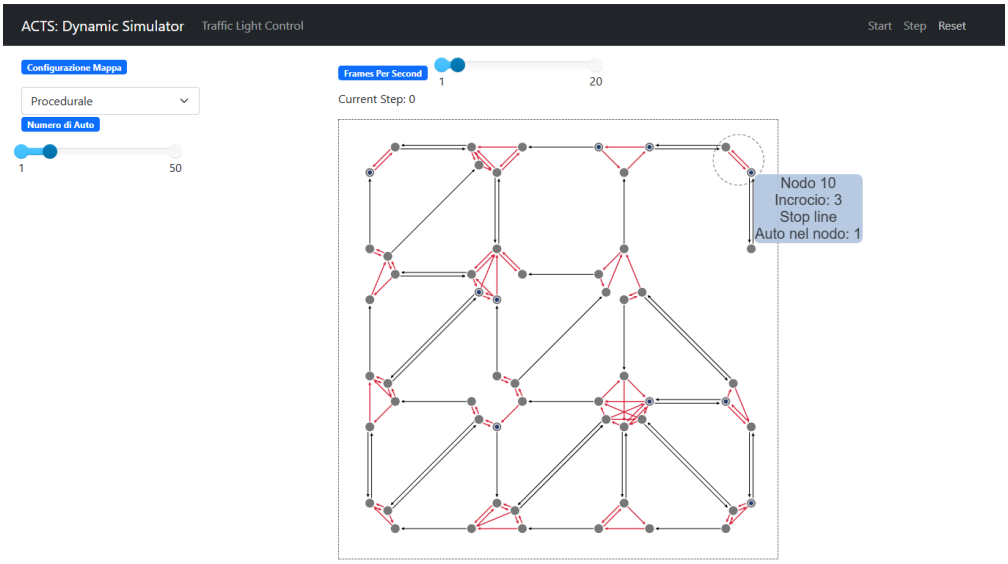

Figure 5: Main simulator interface. From the side panel, it is possible to configure the map, number of vehicles, and execution speed (Frames Per Second).

7.2.1 Available Scenarios and Expected Outcomes

The user can validate the soundness of the distributed algorithms by selecting specific scenarios. For each of them, the expected behavior (*Expected Outcome*) is as follows:

- **Random Map (Procedural):** Generates a random city. *Expected outcome:* Vehicles compute valid paths and traffic lights self-organize to relieve traffic without pre-existing manual configurations. An example is shown in Figure 6.

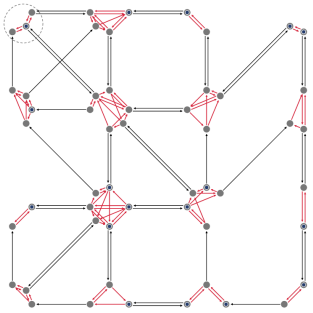

Figure 6: Example of a procedurally generated graph.

- **Basic negotiation:** *Expected outcome:* Demonstrates the base protocol: two traffic flows alternate orderly without collisions at the intersection. Shown in Figure 7.

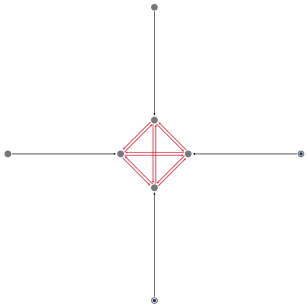

Figure 7: Two groups of cars arrive simultaneously at an intersection.

- **Arrival order:** *Expected outcome:* Demonstrates proper vehicle priority based on arrival. Whoever occupies the intersection entrance first will receive the green light first. Shown in Figure 8.

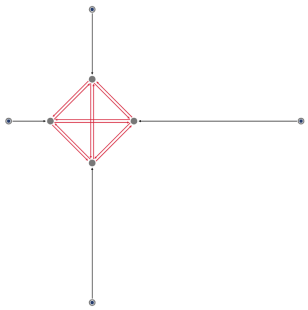

Figure 8: Four cars arrive at the intersection at different times. The crossing order will mirror the arrival order.

- **Fairness:** *Expected outcome:* Fairness verification. Even if one direction exhibits a heavy, continuous flow of vehicles, the algorithm forces yielding green after `MAX_GREEN_TIME`, ensuring that isolated cars on secondary roads do not suffer infinite waiting times (*starvation*). Shown in Figure 9.
- **Traffic wave:** *Expected outcome:* Demonstrates proactive synchronization. The creation of "green waves" will be observed: downstream traffic lights will turn green before vehicles even arrive, thanks to notices sent by upstream traffic lights. Shown in Figure 10.
- **Lamport conflict:** *Expected outcome:* Conflict resolution. Two vehicles arrive at the exact same simulated instant. The system does not deadlock (*deadlock*); one of the two traffic lights yields based on the mathematical priority established by the Lamport Clock and node ID. Shown in Figure 11.

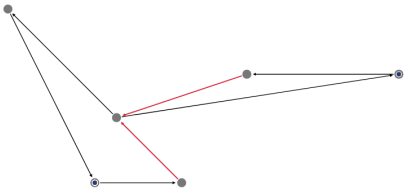

Figure 9: An infinite loop of cars attempts to occupy an intersection indefinitely. An isolated car with lower priority will still manage to cross.

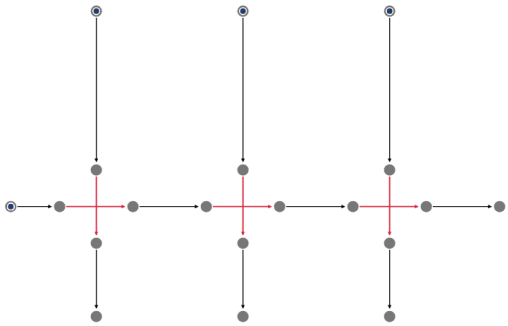

Figure 10: A group of cars crosses 3 different intersections; their passage attempts to establish a green wave even involving busy intersections.

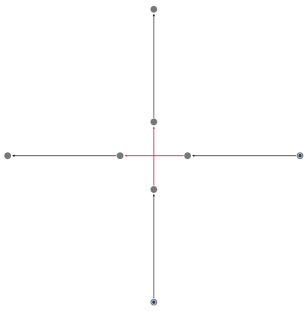

Figure 11: Two cars try to occupy an intersection at the same time; the perfect tie between scores leads to tie-breaking with Lamport clock and agent ID.

7.3 Real-Time Monitoring

During execution, the user can visually inspect the internal metrics of each agent simply by hovering the mouse cursor over the road grid nodes (Figure 12).

Expected outcome of the interaction: A tooltip will appear on screen showing real-time telemetry data: the current traffic light phase (e.g., GREEN, RED, FLASHING_YELLOW), the number of queued cars, cumulative waiting seconds, and the topological ID of the node.

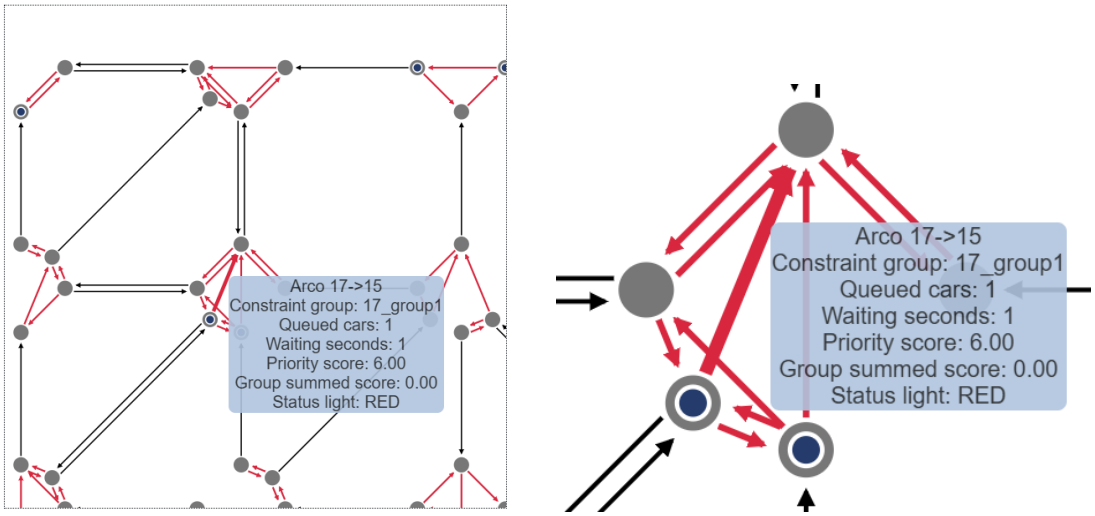

Figure 12: Overview of the network topology (left) and detail of real-time metrics for a single traffic light node exposed via tooltip (right).

7.4 Control Panel and Fault Injection

To test the system's *Fault Tolerance*, the user has access to a separate service dashboard, accessible via the **Traffic Light Control** link in the top navigation bar.

This interface (Figure 13) lists all active intersections and allows manual *override* intervention.

Figure 13: Traffic control panel (Traffic Light Control). Allows inspecting the state of intersections and injecting faults manually (e.g., disabling a node via the "Turn off" button).

7.5 Diagnostic CLI Tool

For an in-depth analysis of distributed network traffic, a dedicated command-line tool is available.

Usage instructions: Open a new terminal and launch the diagnostic script:

```
python src/tests/monitor_traffic.py
```

Expected outcome: The CLI will subscribe directly to the Redis broker, intercepting and printing to the screen in real time the asynchronous stream of JSON packets exchanged by agents. This tool allows developers to *debug* the distributed consensus protocol (verifying the correct transmission of `REQUEST_GREEN`, `ALLOW_GREEN`, and `HEALTH_CHECK`) without having to rely exclusively on graphical rendering.

8 Self-Evaluation

As required by the project specifications, this section contains individual self-evaluations. Each group member analyzes their role, the work carried out, and provides a critical assessment of the strengths and actual limitations of the final product.

8.1 Self-Evaluation: Federico Morsucci

Role within the team

The general system logic design was conducted in close synergy with Cristian. The approach adopted for resolving criticalities and bugs was highly collaborative, based on continuous discussion aimed at identifying optimal architectural solutions. From an implementation standpoint, my contribution focused primarily on generating the network topology and managing the lifecycle of vehicles (*VehicleAgent*). In parallel, I handled the refinement of internal decision-making logic for traffic light agents, ensuring proper interaction and synchronization of distributed nodes within the spatial environment.

Strengths of the product

- **Simulation flexibility:** The main strength of the completed work lies in having built a highly flexible and scalable environment. The system allows dynamically generating different topologies, rapidly testing distributed logics, and observing real-time network behavior as vehicular load varies. This versatility significantly accelerated the iteration and empirical validation phases of the algorithms.

Weaknesses and limitations

- **The Mesa framework compromise:** Adopting the Mesa library provided a major initial advantage for managing simulated space and time. However, integrating purely asynchronous network mechanisms within a discrete-*tick*-governed architecture significantly increased complexity, forcing the use of specific *workarounds*. While the overall balance of its use remains positive, this paradigmatic divergence occasionally made the implementation cumbersome.
- **Code maintainability:** Having prioritized the stability and correctness of distributed communication protocols, formal code optimization was partially sidelined. To ensure high long-term maintainability, several sections of the *codebase* would require a *refactoring* effort. Despite this engineering limitation, the underlying distributed concepts and architectural logics remain rigorous and sound.

Final assessment

Developing this project proved to be a highly valuable learning experience. Witnessing the system's evolution and watching coordinated behavior emerge was deeply rewarding.

At the same time, the work highlighted the practical complexities of distributed software development: network asynchrony introduces concurrency anomalies and message timing issues that are difficult to anticipate during design. Overall, it served as an excellent testbed for consolidating theoretical fundamentals while grappling with the real challenges of decentralized systems.

8.2 Self-Evaluation: Cristian Morbidelli

Role within the team

We conceptualized the project's overall structure together, but on a practical level, I focused heavily on translating distributed rules into physical behaviors on the map. I worked extensively on graph modeling using *NetworkX* and on the agent protocol and its implementation, involving *Failsafe* mechanisms and ensuring that nodes reacted correctly to the states and failures of their neighbors.

Strengths of the product

- **Visible fault resilience:** The *Heartbeat* and timeout mechanism works very responsively. I am satisfied with how forcefully disabling an intersection from the control panel allows one to watch the network reorganize in real time: neighboring nodes isolate the failed node by switching to flashing yellow, and vehicles reroute.
- **Agent protocol modularity:** The clear separation between graph topology and agent communication logic makes the architecture highly extensible: one can modify the map or add new intersection types without rewriting synchronization and *Failsafe* rules.

Weaknesses and limitations

- **Broker bottleneck:** For implementation convenience, we relied on a single Redis instance. In practice, the logical architecture of the traffic lights is decentralized (there is no master), but the network infrastructure has a *Single Point of Failure*. If Redis crashes, the entire system goes silent.
- **GUI scalability:** The Tornado server and D3.js rendering provided by Mesa suffer significant slowdowns when attempting to scale the simulation to very large grids with hundreds of vehicles, limiting the stress tests we could perform on the network infrastructure.

Final assessment

It was an extremely "hands-on" project. The most interesting part for me was implementing failure responses: forcing the network to remain stable when part of the system suddenly disappears. Despite the limitations of the physical simulator, the distributed

core of the project respects all required mutual exclusion and liveness properties, proving that a *peer-to-peer* approach to traffic management is complex but fully viable.

9 Future Works

While demonstrating the validity of the decentralized approach for traffic management, the ACTS project lends itself to several developments. Future development directions focus on integrating real-world operational scenarios, infrastructure optimization, and overcoming the limitations imposed by academic simulation frameworks.

9.1 Architectural and Operational Evolutions

- **Emergency Vehicle Management (Priority-based Routing):** A fundamental extension would be the implementation of a dynamic priority hierarchy. In the presence of emergency vehicles (e.g., ambulances), traffic light nodes should trigger a global preemption request. Agents along the path of the emergency vehicle should force a green light, coordinating with neighboring nodes to clear the emergency corridor in real time. This would require an evolution of the negotiation protocol, transitioning from a logic of *fairness* to a logic of *absolute urgency*.

- **Transition to a Real-World Execution Environment:** To overcome the limits imposed by the *Mesa* framework, the next step would be migrating the system from a simulated environment to a *hardware deployment* environment. The developed distributed logic, being independent of the simulation engine, is already ready to be ported to *Edge* nodes (e.g., Raspberry Pi or industrial microcontrollers installed at intersections). This would allow testing the system under conditions of real network latency, jitter, and physical link instability.

- **Overcoming the Single Point of Failure (SPOF):** The current reliance on a single Redis instance represents an infrastructural point of weakness. A necessary evolution would be the transition toward a high-availability Redis *cluster* or the adoption of a distributed messaging protocol (e.g., distributed MQTT or architectures based on federated broker nodes), eliminating all forms of centralization.

9.2 Optimization and Advanced Features

- **Security and Authentication (Secure Messaging):** To make the system robust in open urban contexts, integrating a *Secure Messaging* protocol would be necessary. We would implement digital signature mechanisms based on asymmetric cryptography: each `REQUEST_GREEN` message would be signed by the sender node, ensuring data authenticity and preventing *spoofing* attacks or malicious message injections (Byzantine behaviors).

- **Binary Serialization Protocols:** The current use of JSON over TCP, while excellent for the *debugging* and analysis phase, is bandwidth-heavy. Adopting binary serialization protocols (e.g., Protobuf or gRPC) would drastically reduce packet size, improving performance in dense traffic scenarios where the number of messages per unit of time grows exponentially.

- **Dynamic Topology Adaptability:** Currently, the map is static after initialization. A future direction of great interest would be real-time topology management: in the event of accidents or unexpected road closures, the system should be able to propagate *topology update* events across the entire network, forcing a dynamic recalculation of graph weights and vehicular *routing* strategies in a fully asynchronous manner.

References

[1] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[2] Project Mesa Contributors. Mesa: Agent-based modeling in python. `https://mesa.readthedocs.io/`, 2026. Accessed: 2026-07-20.

[3] Redis Ltd. Redis pub/sub documentation. `https://redis.io/docs/latest/develop/interact/pubsub/`, 2026. Accessed: 2026-07-20.

[4] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Pearson Prentice Hall, 2nd edition, 2007.

[5] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.