

Smart roundabout controller

Final Report for the Distributed Systems Course [2025-2026]

Daniele D'Ugo

`daniele.dugo@studio.unibo.it`

September 18, 2026

The project consists in a simulation of a Cyber-Physical System, where some autonomous cars have to coordinate with each other to cross a roundabout, each from its starting point to its destination. Moreover, a central controller is available to give them more efficient commands, trying to maximize the intersection's throughput while always maintaining safety as the first priority. In fact, the main distributed systems challenge concerns the reliability and fault tolerance of the system as a whole: network and performance stability are also important, but failures, assumed to be inevitable, should never result into car accidents. Several measures have been implemented or just theorized in order to prevent deadlocks and starvation, to guarantee liveness, to coordinate together healthy vehicles with unreachable ones and to improve controller's predictions and evaluations in general.

An application has also been produced to visualize the simulation in real time and control it through some commands.

1 Concept

The developed product is a simulation for an hypothetical (and simplified) scenario where some coordination is required to guide a certain number of autonomous cars through an intersection with a roundabout: a Cyber-Physical System. Each vehicle is able to proceed on its own, but a central controller is present to communicate with them, in order to optimize the traffic. The simulation can be visualized through a web application and exposes, through a web service, some commands, for example accessible via terminal scripts or via the application itself.

The simulation has been thought to run endlessly without the need for human interaction; the commands may be used for debugging or for testing particular scenarios (pause/resume, inject chaos by changing the state of some vehicles to disconnected from the network, fetch information on them). No user data is thus required to be stored.

Even inside the simulated world, cars are completely autonomous and don't require any human interaction. The central controller only needs to collect from each one some

spatial information (such as position and speed) and store it for a small amount of time (e.g. 1 second), after which data is considered stale and no longer useful.

This scenario could already be considered a distributed system due to its own nature, where the cars are nodes that, in order to navigate safely, need to at least gather information on each other's position. Adding a central controller, then, opens up for more possibilities to improve the efficiency of the system, while making sure that it's safe for its participants at all times: knowing each one's position and destination and being able to adjust the speed of each one, it can optimize their interleaving order and make them run under more sustainable conditions. For example, braking too hard, with time, wears some components such as the tyres; on the other hand, avoiding higher speeds when not needed will also save some fuel, besides carrying fewer risks in general.

2 Requirements Elicitation and Analysis

In general terms, the functional requirements of the project concern the building of a simulated environment, together with an application to visualize it and some commands to manage it.

- 1 — The system shall comprise a roundabout, surrounded by a certain number of straight roads (e.g. 4).
- 1.1 — The roundabout has a single lane and vehicles inside move counter-clockwise.
- 1.2 — A road is made of two lanes, for opposite directions; vehicles ride on their respective right side.
- 2 — The system shall comprise a certain amount of cars (in a reasonable number, e.g. 30).
- 2.1 — From a top-down perspective, a car's surface covers a rectangle, both visually and physically (e.g. for collisions).
- 2.2 — A car is able to move in space, at a speed which can be tuned between certain minimum and maximum values by changing its acceleration. The physical model thus comprehends simple uniform motion and uniformly accelerated motion, both linear (for a straight road) and circular (around the roundabout). Acceleration also ranges from a negative minimum (for braking) to a positive maximum; the minimum amount (i.e. strongest braking) should be used as little as possible, to reduce the car's wear.
- 2.3 — Each car is assigned an entry road and an exit road: it will come from the former towards the central roundabout, ride some distance inside it and then exit from the latter. It can spawn on any point on the entry road and despawn once it's ridden for some distance on the exit road. The distance ridden inside the roundabout must be strictly greater than zero and no more than a full lap,

meaning that having the same entry and exit road implicates doing exactly one lap.

- 2.4 — Each car is able to proceed autonomously.
- 2.5 — Each car is provided with a local vision, allowing it to perceive others nearby (in reality, it could be implemented through infra-red, for example).
- 3 — The system shall comprise one controller.
- 3.1 — The controller can receive spatial data from vehicles.
- 3.2 — The controller can send commands to vehicles, to alter their acceleration.
- 4 — The system shall run a simulation, where vehicles move from each one's start to end point while communicating with the controller.
- 4.1 — In the simulation, vehicles can be in one of three states: normal (correctly communicating with the controller and following its orders); failsafe (correctly communicating towards the controller, but not receiving, or receiving but not following, its orders); disconnected (completely unable to either send or receive data to or from the controller).
- 4.2 — The simulation constantly checks for collisions between cars, in order to report them.
- 5 — The system shall offer ways to both visualize and give some commands to the simulation.
- 5.1 — The system allows to visualize the simulation in real time, together with additional data on the vehicles, such as current speed and acceleration.
- 5.2 — The system allows to send commands to control the simulation including: pausing and resuming the simulation, changing the state of one or more vehicles.

In the project, we identify the **roads** by their **index**: 0 represents the road pointing straight to the right of the roundabout; other road's indexes follow a counter-clockwise order. For instance, with 4 roads, 0 is to the right, 1 up, 2 left, 3 down.

Non-functional requirements are instead related to distributed systems matters:

- 6.1 — **Fault tolerance** of the entire system: the controller or the network may even experience malfunctions, but the safety of vehicles should be maintained at all times, to prevent accidents.
- 6.2 — **Openness** and **interoperability**: the system should aim to be open and interoperable with different implementations (for example, from different car brands, using different technology).

Implementation requirements don't impose any particular technology, but rather an effort to employ two development methodologies strongly encouraged in the course's project requirements and which I personally have little or no experience with:

- 7.1 — **model-first** approach
- 7.2 — **unit-testing**, possibly before writing the code itself (**test-driven** approach)

2.1 Relevant Distributed System Features

2.1.1 Transparency

Despite the general advantages of a higher level of abstraction for the user, **transparency** is only required in a few forms. The main one is **scalability transparency**, since the number of vehicles will vary over time (also refer to 2.1.3). **Replication**, **migration** and **performance transparency**, instead, aren't strictly required, but are always a good feature for the central controller. Lastly, **location transparency** is necessary, since vehicles are moving all the time.

On the other hand, **concurrency transparency** isn't relevant, since there's no shared resource to access, and **failure transparency** isn't granted, since failures are actually taken in great consideration in the project requirements themselves and one component needs to know about others' failures.

2.1.2 Dependability

Fault tolerance is the main focus of the project, since **safety** should be granted at all times (requirement 6.1, 2). Even with network issues, the system should provide **reliability** globally.

2.1.3 Scalability

As already mentioned, the number of vehicles will vary over time; however, it should be noted that the roundabout's surroundings can only contain a limited amount of cars, hence the actual number of involved nodes at the same time isn't expected to grow a lot. For a different, larger scenario, for example trying to coordinate traffic over a wider area, this may become a more urgent issue.

2.1.4 Security

Security would be very important in this scenario, in the real world, since giving false information would mislead the controller's calculations on one side and result in dangerous instructions for the vehicles on the other: thus, it would be particularly important to guarantee data **integrity** and use **authentication** to prevent the injection of false data.

On one hand, in fact, wrong commands could lead to accidents; on the other hand, an overflowing of inexistent cars' positions could lead the controller to block the entire system.

2.1.5 Resource sharing

No resource has been planned to be shared.

2.1.6 Openness, interoperability, heterogeneity of components

As stated in constraint 6.2 (2), the system should be open to potentially welcome external vehicles' implementations, for example even with different languages and technologies.

2.1.7 Evolvability, maintainability

The system doesn't require frequent updates, unless there are changes to the road or to traffic rules, but it may be extended in the future with additional features or performance optimizations. The logs could be useful to collect and analyze, for example, in case of an accident.

2.1.8 Performance, concurrency, computation/communication efficiency, bandwidth

Since vehicles are constantly moving, possibly even at high speeds, it's important to pose a limit to the latency of the communication of commands from the controller to each one: a significant distance can be ridden in less than a second.

Regarding computational performance and network load, because the system isn't expected to grow significantly (as mentioned in 2.1.3), there shouldn't be a need for particularly powerful expensive servers and network interfaces to sustain enough vehicles.

2.1.9 Economy, costs

There shouldn't be any particular constraints to limit economical costs: once servers and network interfaces have been set up, in a real life scenario, there don't seem to be any measures which could be employed in the development to heavily impact the budget (also refer to scalability, 2.1.3, for the expected system's dimensions).

3 Design

3.1 Architecture

The architecture is **event-driven**, revolving around a **message** oriented **broker**, through a **publish-subscribe** mechanism. Under normal conditions, there's a centralized controller to orchestrate traffic; however, each vehicle is also provided with an autonomous and fully functioning logic (*failsafe mode*), in order to work even under network unavailability. Finally, a **server**, subscribed to the same message broker, employs a REST API to serve the simulation visualization to a client and to receive configuration commands.

This architecture type addresses many of the distributed systems features discussed in the requirements.

First of all, the message based broker already provides naturally many forms of **transparency** (2.1.1), since everyone has to only communicate with this, without having to know many details about the other nodes, not even their number (**scalability transparency**), or having to share additional information with them.

Moreover, a broker, using a standard protocol for messages, makes it straightforward, without any additional modification or configuration needed, to: change the number of nodes (**scalability**, 2.1.3); make modifications and improvements in a modular way, to a single infrastructure component (evolvability, 2.1.7); integrate different technologies, as long as they use the same communication protocols (openness, 2.1.6).

Implementing an autonomous logic in each car was fundamental: it's the only way to guarantee the required fault tolerance (2, 2.1.2). In fact, even if the controller was perfectly implemented to be always available, there could always be network issues, and a car accident can't be accepted. Furthermore, one should also expect that some vehicles may be completely unreachable, creating problems the controller can't always arginate with its counter-measures (refer to controller's behaviour, 3.5). Finally, even in normal conditions, a local logic, without any network latency, constitutes an additional safety measure.

The web server doesn't have any other notable characteristics, since it's expected to only be used by one, or few, people to visualize the simulation, thus without a heavy load; even in a real life scenario, a similar solution may be used by just a few system's maintainers for monitoring purposes.

3.2 Infrastructure

The infrastructure follows simply from the architecture (3.1), with these components:

- A car (replicated in multiple instances), comprising fictional sensors (sensing data), actuators (applying instructions) and its own logic.
- One controller.
- One message broker.
- One simulation service: subscribed to the same broker, it handles some tasks about the simulation itself, most of all detecting collisions between vehicles.
- One web-viewer.

The system uses Docker containerization to simulate a geographically distributed environment. Each vehicle and each other component runs as an isolated process in its own container, sharing only the broker's virtual network. So, every communication happens through the broker, except the REST calls from the simulation's human spectator to the web server.

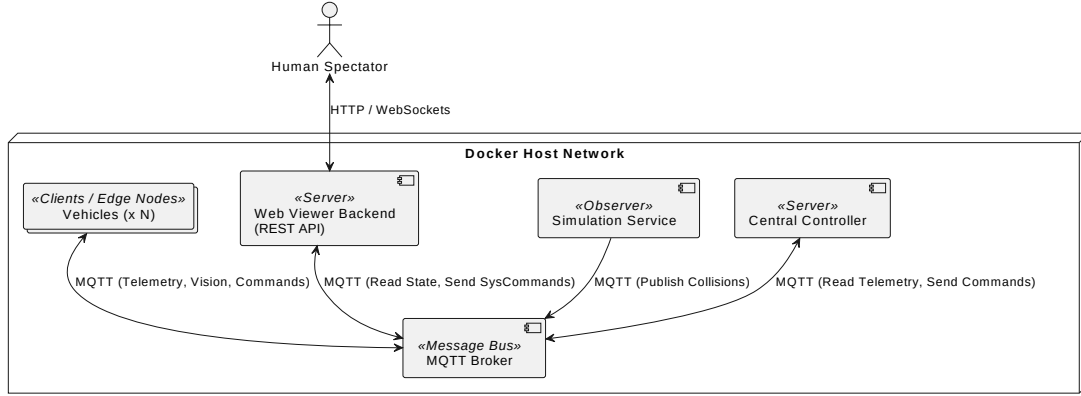


Figure 1: The system's components and their communication paths.

3.3 Modelling

Domain entities are mostly constituted by the vehicles: the entity defined as **vehicle position** only contains spatial data (position and speed); the **vehicle** entity contains the same spatial data plus some additional information, such as acceleration, entry and exit road (to the roundabout) and variables for displaying. Furthermore, for both vehicles and positions, a **navigation state** is defined for convenience: **approaching** when going towards the roundabout, **inside** when driving around it, **exiting** when going away. Notice how a **vehicle position** reflects our model of what a car's local vision can perceive: infra-red sensors would be able to quickly read another object's speed and position.

In addition, there are entities for **controller commands** (issued to vehicles) and **system commands** (configuration controls issued by the viewer).

Each vehicle maintains:

- its **local state**, as a **vehicle**;
- a **vehicle position** for each car in its vision's range: a subset of the global state.

The controller maintains:

- **active_vehicles**: a dictionary of healthy **vehicles**, transmitted directly by themselves;
- **precedence_queue**: a queue to manage the order of arrival at the roundabout's entrance;
- **ghosts**: estimated **vehicle positions** of disconnected vehicles (reconstructed through the healthy ones' crowdsensing).

All data on the controller's node is kept up-to-date, i.e. old entries (older than a certain threshold) are periodically purged, hence its never too different from the real-time data, uniquely known by each vehicle alone.

All commands are exchanged over the network and used immediately, never stored. Exchanged messages reflect the entities, and how they're perceived by each component:

- a periodical message for each healthy **vehicle**'s telemetry;
- a periodical message for each **vehicle** (even the disconnected ones): the controller doesn't utilize this, since this data wouldn't be available in real life, but the simulation does, in order to display the real state to the viewer;
- a periodical message for the objects in the **local vision** of each healthy vehicle;
- periodical messages for **controller commands**, indicating what acceleration the car should use;
- periodical **controller** messages to share its **state** (for debugging), containing its managed entities;
- messages for **system commands** (dubbed *sysctrl*), from the user (over REST API) to the web viewer (which will then communicate them to the correct recipient);
- messages for **system commands** (*sysctrl* MQTT messages), from the web viewer (after receiving one from the viewer), either towards individual vehicles or towards all at the same time in broadcast;
- messages for detected *sysctrl* collisions, from the simulation component, to be displayed to the viewer;
- a message sent by a vehicle to communicate that it has *sysctrl* reset its position (about reset in 3.5.3), for displaying purposes;

Related to the messages, the main events can be identified as:

- *TelemetryUpdated*: a vehicle shares its kinematic state;
- *VisionUpdated*: a vehicle shares the objects detected through its local vision;
- *CommandIssued*: the controller has evaluated the state and instructs a vehicle to accelerate or brake;
- *CollisionDetected*: the simulation service identifies a car collision;
- *SystemStateChanged*: the viewer user issues a system command;
- *HeartbeatMissed/Timeout*: the timeout for lack of messages triggers a vehicle to transition to a failsafe state.

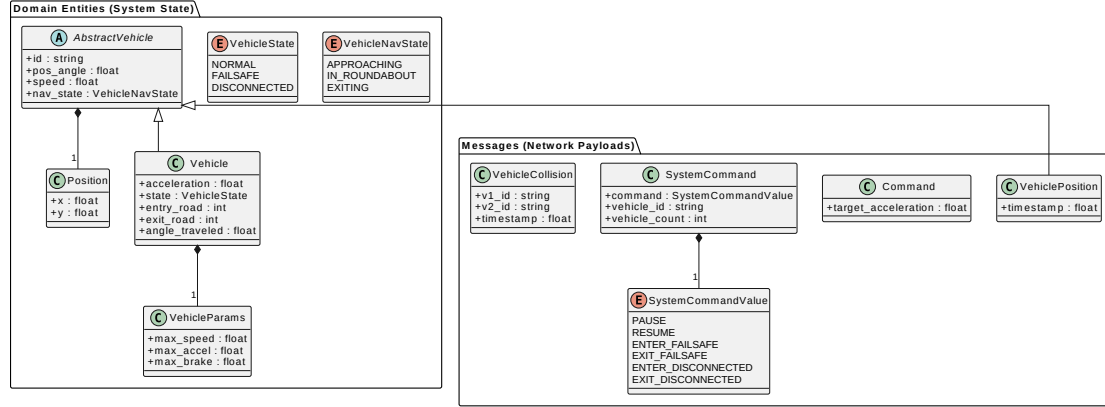


Figure 2: Domain entities, as classes. Vehicles are used to represent the system’s state, whereas other entities are transmitted as messages payloads.

3.4 Interaction

All communications happen through the message broker, except the REST calls between front-end and back-end of the visualizer web service.

Each **vehicle** periodically shares his data through the messages seen in section 3.3, with a constant rate:

- its **telemetry**, if it’s in a healthy state, for the controller;
- its **telemetry**, in any state, for simulation management and visualization purposes;
- its **local vision**, if it’s in a healthy state, for the controller;

The **controller**, using the last vehicles’ messages collected asynchronously, periodically issues **commands** back to the responding vehicles.

The **simulation service** shares a collision when it’s detected, on a dedicated message channel — read by the web service for visualization.

The **web server** displays vehicles, vehicles’ states and controller’s state, all fetched periodically throught the broker. Moreover, upon receiving a system command from the viewer, it sends the proper message (through the broker) to the correct vehicles.

3.5 Behaviour

In general, vehicles represent the “truth” of the system, since they are the ones physically occupying a certain position and moving around, regardless of the state of the network; to them, the controller is an external service which, in normal conditions, they consult to receive the most efficient, safe driving command. Finally, the simulation and visualization services are mostly mere observers — with the visualizer that can forward system commands.

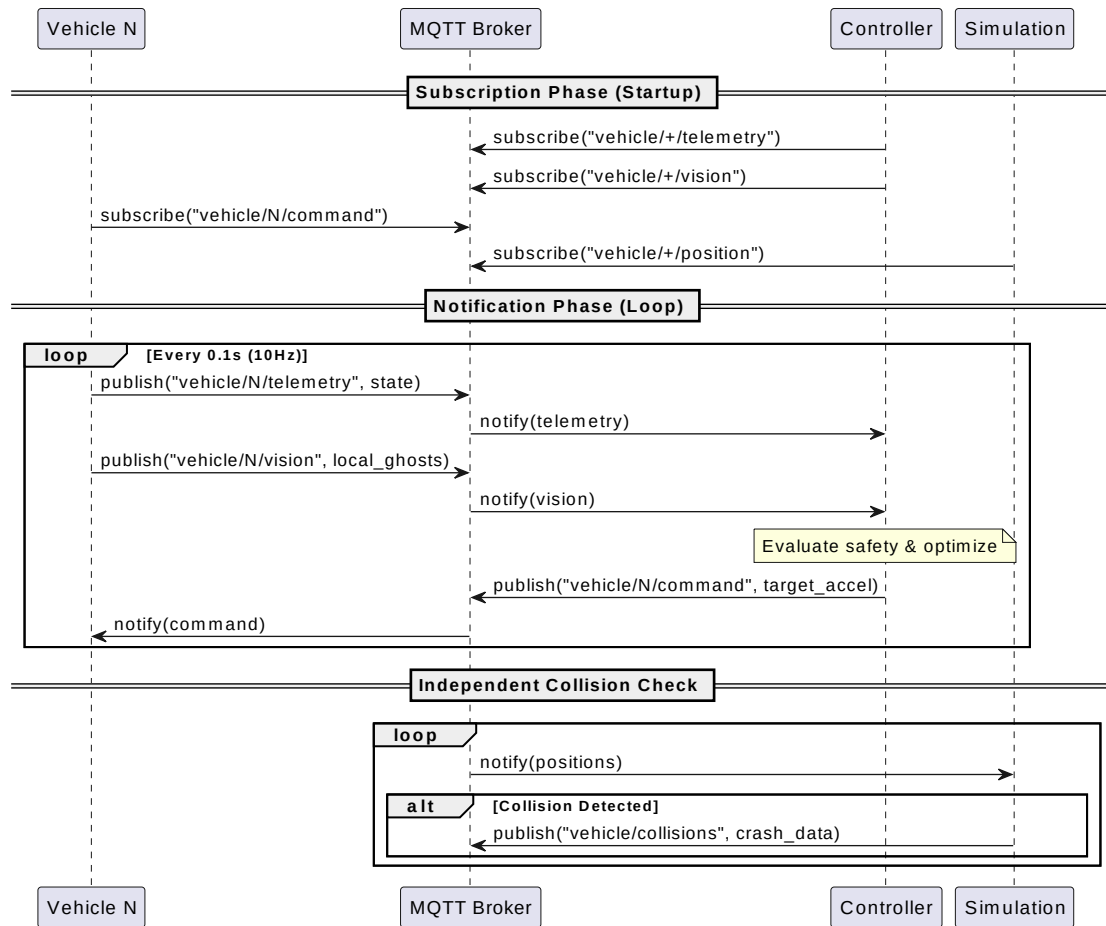


Figure 3: Sequence of messages exchanged between the system components.

3.5.1 Controller

Starting from the **controller**, it continuously listens to vehicles' messages (for their telemetry and their local vision) and, at the same time, it periodically evaluates and issues commands, with a constant rate (currently 10 times per second).

Each received vehicle telemetry is saved together with a timestamp, in order to only keep the most recent data and purge entries older than a certain threshold (currently 0.5 seconds): these are called “**active vehicles**”. Moreover, before each evaluation cycle, the position at the current time is estimated for each one, by moving them by the distance they would have ridden, in their current state, in the time interval since their communication: in this way, not only are the positions presumably more accurate, but they also all refer to the same instant.

The vehicles in their local vision, called “**ghosts**”, are instead all collected together, since they're just positions and can't be merged directly even if they represent the same one; to reduce their number, a more complex technique is needed. Basically, each ghost is first moved to its predicted current position (like for *active vehicles*), then checked for collisions against the set of each *active* one: in case of a match, it's discarded, likely referring to the same one; otherwise, it's saved as a distinct vehicle and will also be checked for collisions with other *ghosts*, to avoid to repeat them.

Since entry and exit road aren't known for ghosts, additional precautions are taken to further optimize the process. Entry isn't actually important: it's obvious for an approaching car and irrelevant for an exiting one; for one inside the roundabout, however, future developments could use it to make some predictions (e.g. with the assumption that they won't make more than one lap, from requirement 2.3, in 2). The exit of a vehicle inside is instead very useful to optimize the entry of others. To grant the safety of the subsequent calculations, the worst options are assumed for a vehicle inside and assigned to each ghost: the last passed road is picked as both entry and exit.

Prediction of vehicles' current positions, ghosts merging and out-of-date purging (for both vehicles and ghosts) are performed periodically before each loop of commands evaluation. It's important to remark that keeping the number of total vehicles (active plus ghosts) as low as possible is relevant for performance, given the complexity: an evaluation has quadratic complexity in the number of total vehicles and ghosts themselves may be the squared number of existing vehicles (if each one sees each other one once), possibly repeated if local vision messages are collected repeatedly. It should also be noted that the more vehicles are disconnected from the network, the less reliable this data will be, having more “invisible” nodes than those contributing to the crowdsensing. In fact, data can only be provided by vehicles able to publish to the network (whether they're responding to commands or driving in failsafe mode), but not by those which are completely disconnected.

The **evaluation** phase runs a sequence of checks for each active vehicle v_1 : in a single loop, v_1 will be compared with each other active or ghost vehicle v_2 — but only active vehicles will receive the resulting commands.

The general traffic principles for the controller are the following:

- a vehicle, in any navigation state, should monitor the one in front to avoid to crash into it;
- approaching vehicles gain a priority level as soon as they get close (within a threshold of a few meters) to the roundabout entry (aim to be FIFO, without being too rigid);
- vehicles inside generally have the priority;
- given the priority queue of approaching ones, a v_1 inside the roundabout should try to slow down and yield to an entering v_2 , if v_2 has a higher priority, but only if this can be done *safely*. In the same way, v_2 , even if it has higher priority on v_1 , shouldn't enter if this would put v_1 in a dangerous situation.

Controller's evaluations thus try to apply any possible optimization, as long as these principles are always respected. To search for the highest safe acceleration, for each v_1 , the command is initially set to the maximum possible value; each subsequent check against every v_2 can only lower it.

First of all, the **safety distance** for v_1 from v_2 in front is defined, assuming they both started braking immediately, as the distance ridden by v_1 before it stops minus the one ridden by v_2 , plus the reaction distance of v_1 , plus an optional margin (so that the cars won't touch each other once they will have stopped). Reaction distance, for v_1 , is itself defined as the distance ridden, with current speed and acceleration, in the time before v_1 realizes it needs to brake and starts slowing down.

$$safe_dist = (stop_dist_{v_1} - stop_dist_{v_2}) + react_dist_{v_1} + safe_margin$$

As optimal parameters, one should use the maximum braking acceleration for v_2 , in case it has to brake for emergency, and a light deceleration for v_1 , so that it's able and ready to stop at any time with minimal wear on the mechanical components (requirement 2). Reaction time for the controller should be equal to the data staling timer and the network timeout, because v_1 would have ridden more distance in that interval, potentially getting too close to v_2 ; on the other hand, a self-controlled vehicle can have a negligible reaction time. Lastly, the safety margin can be set to a few meters, for example the length of one car.

The main priority for the controller is to respect this constraint at all times: v_1 can't accelerate if it doesn't have enough braking margin. Taking advantage of the additional level of protection provided by the car's own emergency braking system, the controller only sends a soft braking command when the distance $dist$ between v_1 and v_2 is below $safe_dist$: if the message was to arrive too late to v_1 , it would still start to brake on its own when $dist$ gets below the lower safety distance calculated locally, i.e. using a lower reaction time.

After having made sure that v_1 doesn't have to brake before a v_2 in front, if it is close

to or inside the roundabout, the controller has to handle the most critical points: the entrances.

When v_1 is about to enter, it will check for all vehicles already inside. Those that will exit before its entrance can be safely ignored. For every other v_2 , it will be checked if v_1 can coordinate safely with it, i.e. if v_1 can either pass before or after while maintaining the proper safety distance: if v_1 passes first, it shouldn't go so slower than v_2 such that v_2 will hit it soon after entering; if v_1 passes last, v_1 shouldn't go so fast that it will hit v_2 after entering. The checks are performed for different values for v_1 's acceleration, positive if possible, otherwise null (i.e. constant speed) or negative (soft or hard braking) in the worst cases.

The v_2 inside itself is also under examination, in case it has to yield (to someone with higher priority) or in case v_1 is close to the entrance. In both situations, v_2 will try to facilitate the entrance of v_1 by slowing down if needed, but only if it can be done safely — otherwise, recall that who's inside has the priority. Nonetheless, to safeguard the traffic flow's liveness, vehicles inside never go below a certain speed threshold, unless the safety distance has been violated, with respect to either the vehicle in front or the one about to enter.

Vehicles in a *failsafe* or *disconnected* state are also given higher precedence at the intersections, i.e. a responding vehicle will try to yield to them (again, safely). The first reason is to make them pass and get rid of them as soon as possible, since they will drive slower and less optimally. The second reason is to avoid **starvation** at all: vehicles in failsafe mode are more “timid” and would risk to never be able to enter, especially if they are only one or a few (refer to 3.5.2 for vehicle logic).

Before all, however, in order to make the single loop of commands evaluations more optimal, all vehicles are **sorted** through a simple heuristic, trying to put the ones with a higher priority first: exiting ones, then inside ones, then approaching ones; approaching ones come first the closer they are to the center, while exiting ones come first the further they are; inside ones are picked in clock-wise order.

In this way, before evaluating for v_1 , the speed of a v_2 in front is always already known, since it will have been processed first: this makes the safety distance estimates more correct. The only exception, not taken into consideration, regards the “junction point” for the ones inside: since they are taken in clock-wise order starting from a random point, the first picked vehicle will be evaluated before the one in front of it. Inside, on the other hand, vehicles come before approaching ones because they generally have the priority. Once in the exiting state, instead, they have already passed the intersection and don't have many possible conflict, but should only care about who's in front.

One final controller's feature is aimed at preventing a deadlock inside the roundabout, in a case where all vehicles have halted but no one can resume because they're all too close to the one to their respective front: according the described logic, in fact, such a vehicle wouldn't accelerate until it has enough margin (safety distance). Hence, if all inside vehicles have been stationary for a certain amount of time (e.g. 2 seconds), the controller will trigger a command to accelerate to all; if there are some that have more

margin than others, it will try to only send the command to them. In the worst case, all vehicles should proceed extremely slowly until someone exits, continuously alternating an acceleration and an emergency brake soon after. In a future implementation, all vehicles in a line could start accelerating at the same time and same pace, without colliding, although one should then be sure that there won't be any abrupt or unexpected moves, or they won't be ready to brake with that small safety margin.

In such a deadlock state, the controller won't allow any other vehicles to enter until a certain number of those inside has exited.

Another possible option, for the controller, would be to count inside vehicles and urgently stop others from entering when there are too many.

Anyway, there are already other measures trying to prevent such situations, seeing that inside vehicles try not to go slower than a certain speed and generally have higher priority.

Another possible reason of blockage is when an inside vehicle v_2 has stopped to yield to an entering vehicle v_1 , but they're both too close to the entrance and v_1 has stopped too: in this case, an ad hoc instruction will tell v_1 to accelerate (this is in the vehicle's *failsafe mode* as well).

Knowing the controller's functioning mechanism, it's now possible to see that its complexity is approximately quadratic in the number of total vehicles. Improving estimations through a higher evaluation frequency would thus come at a higher computational cost, possibly even made worse by an higher number of vehicle nodes, so careful examinations would be required first.

3.5.2 Vehicle

Each vehicle repeatedly (currently 10 times per second) runs a procedure: take a decision (on the new acceleration to apply), apply it, update the own position by physically moving according to current speed, direction and destination. In parallel, it also listens for commands from controller and web server and publishes its data.

An actuation loop starts with a local check for an **emergency** command; if one's not needed, the vehicle uses the last received controller's command if available and not too old, otherwise, assuming that there's a network problem, it goes in **failsafe** mode and runs its own evaluation logic. After having taken and applied the decision, it will try in all cases to share its updated position with the controller: if the connection had been lost, this is used as a heartbeat, to try to re-establish it; if it's driving in failsafe mode for necessity, it won't execute received commands, but it will still update the controller so it can perform better evaluations for the others.

The checks performed in both **emergency** and **failsafe** evaluations are very similar to the controller's ones (3.5.1).

The **emergency** evaluation only does checks about safety distance, using hard brakes: in the tail-gating case (against a vehicle in the same navigation state) and near the intersection points, whether it's the one approaching or the one inside.

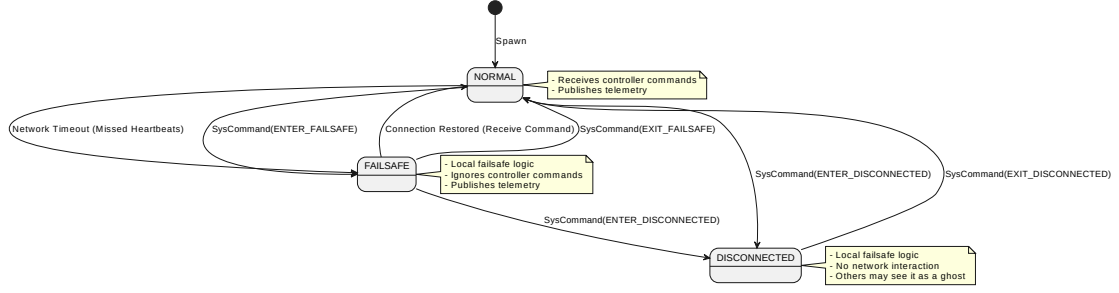


Figure 4: Vehicle states and the transitions between them.

The **failsafe** evaluation, with respect to the controller’s one, has the main differences of imposing a lower maximum speed and of making the vehicle always stop before entering the roundabout. For the intersections, if it’s entering the logic will be similar to the controller one’s, whereas, if it’s inside, it won’t do anything to try to slow down, do optimizations or yield. In fact, regarding the security, the *emergency* mechanism already takes care of all cases of emergency brakes, while, regarding the optimization, the vehicle partly lacks information on the others in order to make the best decisions, partly assumes the others will follow the traffic rules: whether they’re following the controller’s commands or their owns, each logic still aims to only do safe moves.

3.5.3 Simulation

The simulation service only takes care of identifying and publishing any collisions, calculated by looking for pairs of overlapping cars; to avoid an overflowing of messages, there’s a cooldown (e.g. 1 second) for each pair of vehicles, when a collision has been found.

The state of the simulation system is actually kept by the vehicles, each knowing (and publishing) its position.

The number of vehicles remains always the same; when one gets far enough from the roundabout, it resets itself, respawning with new random start and destination.

3.5.4 Web-viewer

The web-viewer service makes it possible for a spectator user to visualize and interact with the system.

The service listens to and saves all vehicles states, controller’s state and simulation’s collisions.

It then exposes API endpoints to read its state and to post controls, to one or more vehicles, to pause and resume the simulation or to force failsafe or disconnected modes.

3.6 Data and Consistency Issues

No data needs to be stored, since everything is used almost immediately and old data is soon considered stale and unusable; on the contrary, for the current tasks, a database would only introduce latency, which would be a problem. Therefore, data is currently only being stored in in-memory dictionaries.

A database may only be useful as an additional service, subscribing to the same message topics in order to collect the system's data for later analysis and debugging (for example, even in case of accidents).

The system uses **eventual consistency**; after all, due to latencies and network delays, every component's view is always behind the physical reality, only known by the cars themselves. Nonetheless, vehicles continuously publish their data, so that a listener will sooner or later fetch the latest state. Therefore, data freshness is prioritized over consistency and purging old data is better than synchronizing it.

On the other hand, the controller tries to improve the consistency of its state with the real one through the prediction mechanism (3.5.1).

3.7 Fault-Tolerance

Accepting the possibility of errors is one of the main concerns of the project; after all, this is a simulation of vehicles, meaning that they are constantly moving and can't afford to stop and wait for the signal to come back or the problems to be fixed. So, when messages don't travel successfully between the controller and a vehicle, they are simply **ignored**: each vehicle also has its own logic (3.5.2), to be able to somehow proceed safely even without the controller's commands; the controller will try its best to provide optimal commands to the remaining healthy ones (3.5.1).

The vehicles' failsafe and emergency modes hence contribute to increase the **reliability** of the system (2.1.2). Because unexpected events may always occur, especially if many vehicles are disconnected from the network, the controller can't have fully reliable data. Moreover, in the real world, there could always be other obstacles on the road, objects, pedestrian, etc.

Timeouts are fundamental to mark vehicles' and controller's data as old and no longer usable. Messages between the two components are used as **heart-beats**: a vehicle is considered healthy as long as it keeps communicating forth and back and, even if it loses the connection, it will return to a normal state as soon as it will be able to share its position and receive a command from the controller.

Finally, some measures have been described with the controller (3.5.1) to prevent **deadlocks** and **starvation**, although they don't cover all possible cases; the case in which most or all vehicles are disconnected from the controller is especially lacking. Regarding starvation, after all, even in the real world it could be a problem in busy situations where a too insecure driver blocks one entry for minutes, letting everyone coming from other directions pass. Regarding deadlocks, the fact that vehicles in failsafe mode tend

to not yield (unless necessary for emergency) when inside the roundabout should be of reassurance (but not guarantee).

Availability and fault tolerance improvements, however, haven't been considered for the **controller**. One possibility would be to deploy more controller nodes to use for *master-slave replication*: one is elected as the working one, while the others only follow to keep their state up to date; in case of failure, a replica will become the new master. This approach wouldn't need to add latency, since the replicas don't need to communicate in normal conditions.

3.8 Availability

Since data freshness is much more important than consistency (analogously to 3.6), **caching** would only be counterproductive for vehicles. Caching could only be useful for the web viewer backend, to reduce the load on the broker at the cost of displaying updates less often to the spectators.

As discussed in 2.1.3, the system isn't expected to scale much in the current conditions. If the simulation was to be expanded to a wider context, like a network of roundabouts or a city, a **load balancer** could be employed to reduce the load of many vehicles on a single controllers, by distributing requests among a cluster of controllers; however, this would introduce the additional challenge of performing global evaluations on separate nodes.

Since more roundabouts are mostly independent (although they could probably even be coordinated with each other), **network partitioning** could be a better option. If the city was divided into multiple zones by geographical partitioning, a controller could be dedicated to each one and thus have a reduced load; with a message topic dedicated to each zone, then, a vehicle would only have to subscribe to right one.

3.9 Security

Security hasn't been considered deeply in the development, for lack of time, despite the observations in 2.1.4.

According to the current design, the system has already been thought to prevent accidents due to wrong commands, thanks to the cars' *local vision* and the *failsafe* and *emergency modes* that can take control in case of need; however, an overflowing of existing cars' positions could still lead the controller to block the entire system.

In this implementation, the docker network is assumed to be secure; future improvements, also to prevent malicious injection of commands, may require authentication through TLS for message exchange and provide a unique cryptographic key to each vehicle. For separation of concerns and privileges, the broker may verify authorization through Access Control Lists, where each role is only authorized to publish on certain message topics:

- *Vehicle N*: can only share its own data and subscribe to commands directed to it;

- *Controller*: can subscribe to all telemetry and publish commands for all;
- *Admin*: can publish system commands.

4 Implementation

The project is built on Docker Compose, where vehicles are deployed as replicas and all services communicate on a virtual network.

As **network protocols**, MQTT was used for messages and HTTP for REST API requests (for the web server).

MQTT fits perfectly the publish-subscribe architecture and is an industry standard for IoT and Cyber-Physical Systems. Moreover, the protocol is specifically designed to work with constrained devices, to grant network reliability. Level 0 Quality of Service has been chosen to grant the lowest latency (performance requirement, 2.1.8), thanks to its lower communication overhead; dropping packets isn't a problem, since having the newest data is always the priority for vehicles and controller, while stale data isn't used. Level 1 has only been used for system commands, which are issued sporadically but whose reception is more important.

All in-transit data is serialized as JSON, which is easily readable for debugging and widely available across different languages.

Regarding the **model-first** constraint (2), it has been respected quite well: the models (in *src/common/models/*) have been defined before writing the code, with a python library enforcing data validation (pydantic). Many of the classes have been defined at the beginning of the development. Some classes have been added later when a new kind of entity, interaction or message was needed, as well as fields of existing classes were modified or new ones created.

4.1 Technological details

Eclipse Mosquitto has been used as the MQTT broker, being a lightweight implementation with native Docker support. Furthermore, it supports TLS authentication, for future development additions.

All services are written in Python (3.12): vehicle, controller, web server, simulation service. The *asyncio* library has been exploited to handle concurrent tasks, for instance for simultaneous listening, publishing and logic loops. The *aiomqtt* (2.5.1) library has been used for MQTT communication. Finally, *pydantic* (2.13.4) has been employed to define domain models, enabling a model-first approach. *pydantic* also automatically handles serialization and deserialization of JSON and enforces data validation.

Regarding the web service, the *FastAPI* (0.141.1) library has been used for the back-end, whereas the front-end is written as a simple HTML5 page, with CSS and JavaScript Canvas.

5 Validation

5.1 Automatic Testing

In the development of this project, an effort was made initially to do **unit-testing** in a **test-driven** approach, as stated in the requirement 2, as a personal new experience. However, the promise wasn't really maintained.

Only in the very first stage, during the development of the first version of the simulated world, without any controller or vehicle logic but only with physical movements and mathematical operations, every time that a functionality was needed, a test was developed first and only then that would have been implemented with a separate function (so that it could be tested alone). Unfortunately, this didn't last long, as the required features, and the related functions, soon became too complex. In the first place, this means that a function becomes too hard to test, with too many possible normal or edge cases, and unexpected ones. The main problem, however, is that it was hard to even know in advance how the controller's or vehicle's logic should have come out at the end, so developing a test for a function that would have been changed or even removed soon after would have taken too much time away from the distributed systems related topics.

That said, just a few unit tests were developed right at the end, to test only a few isolated functionalities.

Proper **end-to-end** test and **deployment automation** were never even taken in consideration at all, given the lack of time, the total absence of personal experience with specific tools and the probably complex general architecture.

For communication and interaction testing, manual tests was deemed enough (5.2).

pytest (9.1.1) has been used for unit testing. All tests (in the *src/tests/* folder) can be run, from the project's root directory, with the command:

```
pytest
```

One should first install the packages required by the tested services (i.e. all except the web viewer), besides *pytest* itself. *src/tests/requirements.txt* contains them, and can be used with:

```
pip install -r src/tests/requirements.txt
```

The developed unit tests for physics (*test_physics.py*) and mathematics (*test_math_utils.py* and *test_roundabout.py*) regard kinematics and calculations: they test, for a few cases, that distances are calculated correctly, for example on a line or on a circle, and that speeds and accelerations (positive or negative) are updated and used correctly, i.e. vehicles ride for the expected distance or the expected amount of time.

Unit tests for the the controller (*test_controller.py*) test the command evaluation in a few cases by creating mock scenarios, verifying that a tail-gating vehicle decelerates to increase the margin from the one on the front, or that a vehicle with a lot of margin on the front keeps accelerating.

Finally, unit tests for the vehicle (*test_vehicle.py*) perform analogous tests to those of the controller, but using its failsafe mode logic, and also check the collision detection.

5.2 Acceptance test

Given the lack of automatic testing, manual testing has clearly been fundamental in the project, to continuously debug it and eventually validate it.

Being the subject a simulated real-world scenario, verifying that everything goes as expected by looking at the visualization is straight-forward, although debugging requires much more effort, by deeply analyzing the code and correlating it with what happens in the simulation, or by checking logs and vehicles and controller statistics. The behaviour one should expect to see is what described in the requirements (2).

For communication and interaction between the services, the simple fact that the project worked was enough: bugs and problems only came after, when the it was the logic to be put under examination.

The developed system commands to force the state of vehicles to failsafe or disconnected mode allowed to test these other failure scenarios.

The ability to pause the simulation was also fundamental, to either stop to check logs and statistics, or to watch from up close the interaction between two cars. Moreover, by pausing, one could change the state of vehicles with precision, for example to set to disconnected one which is already in a dangerous situation, in order to test edge cases.

The simulation has thus been run with different numbers of vehicles (between 10 and 50), to see how the system and the controller's work would change. However, the road configuration has never been changed, although it's possible to easily change the number of roads or the size of the roundabout.

Once run the simulation, tests were made with varying amounts of vehicles into failsafe or disconnected mode.

Some possible evaluation metrics at disposal, available on the simulation page (refer to the user guide, 7), are the number of happened collisions and the total number of spawned vehicles in a certain amount of time, to measure the total throughput of the roundabout. Here follow some example runs; refer to the self-evaluation (8) for a brief analysis.

Each test has been run for 10 minutes:

- 40 vehicles (all in normal state): 309 spawned, 9 collisions
- 40 vehicles (all in normal state): 325 spawned, 7 collisions
- 40 vehicles (30 in failsafe): 335 spawned, 49 collisions
- 40 vehicles (all disconnected): 288 spawned, collisions unavailable (the system went in roundabout deadlock after 8 minutes)

- 40 vehicles (all disconnected): 352 spawned, 72 collisions
- 40 vehicles (10 disconnected): 304 spawned, 37 collisions
- 20 vehicles (all in normal state): 360 spawned, 23 collisions
- 10 vehicles (all in normal state): 221 spawned, 3 collisions

6 Deployment

The system is deployed using docker compose, so a working *Docker Engine* installation and *docker compose v2* are required. Internet access is required to pull the two used images from the web: *python:3.12-slim* and *eclipse-mosquitto:2.1.2-alpine*.

To clone the repository from github:

```
git clone \
  https://github.com/papum20/unibo__projects__distributed-smart-roundabout.git
```

The project is already setup and ready to be immediately deployed; changing configuration variables is optional. One should just be careful to have the docker's used ports free on his computer (currently, only 8080, for the web service).

From linux, it can be **run** using the available bash scripts, which should be launched from the project's root directory. To run it:

```
./src/start.sh
```

Details on the running phase can be found in 7.

To stop docker compose (if necessary, first stop watching the containers logs by pressing *Ctrl+C*):

```
./src/down.sh
```

Remember that any variable change requires to restart the proper containers; to be safe, restart all with *down.sh* followed by *start.sh*.

The only *docker compose* file *src/docker-compose.yml* reads **environment variables** from *src/bash-scripts/clean.env.generated*, but they can be configured from *src/.env* in a more readable way (with spacing): the *src/start.sh* script is a wrapper that cleans *.env* to produce *clean.env.generated*, before deploying with docker compose. Variables are explained through comments in the file and can be better understood by reading this report, but, in general, they involve the following topics:

- ports (for container communication inside the virtual network) and local ports (exposed on the machine);
- message topics names;
- number of vehicle replicas (number of replicated docker containers).

Other variables, related to the implementation itself, can be configured in *common/const.py*: they will be shared among all python containers. The most interesting ones to try out are:

- dimensions for roundabout, roads and visualization area;
- number of roads (*ROUNDABOUT_N_ROADS*);
- additional vehicle safety margin *VEHICLE_SAFETY_MARGIN_M* (refer to 3.5.1);
- car's local vision radius *CAR_VISION_RADIUS_M*;

Other variables concern:

- updates per second, for both vehicles and controller;
- network timeout;
- vehicle dimensions and other technical details;
- roundabout proximity distance: threshold to consider a vehicle as “about to enter”;
- controller's deadlock timer (as seen in 3.5.1);

All variables try to mimic real measures: in meters for distances, seconds for time intervals.

About directories and docker compose structure: in *src/*, there's a separate folder for each service. Except *mosquittoMQTT*, which instead directly uses the standard docker image (with a minimal configuration file), all other services' images copy their own folder into their respective container and run python. The *common/* folder contains common variables and utilities and is copied into all python containers.

7 User Guide

After the docker compose infrastructure has been deployed successfully, the simulation can be visualized, from a browser, on the **page** <http://localhost:8080>. Cars should be seen moving, proceeding endlessly.

A **dashboard** on the left of the page displays:

- in the first row, in order, the total number of spawned vehicles in the simulation, the current number of vehicles, the current number of ghosts identified by the controller;
- on the left, a list with each vehicle's telemetry, sorted alphabetically by their id. For each vehicle, its row contains, in order: the id's first 4 characters; a shortened word for its status (normal, failsafe or disconnected); speed “S” and acceleration

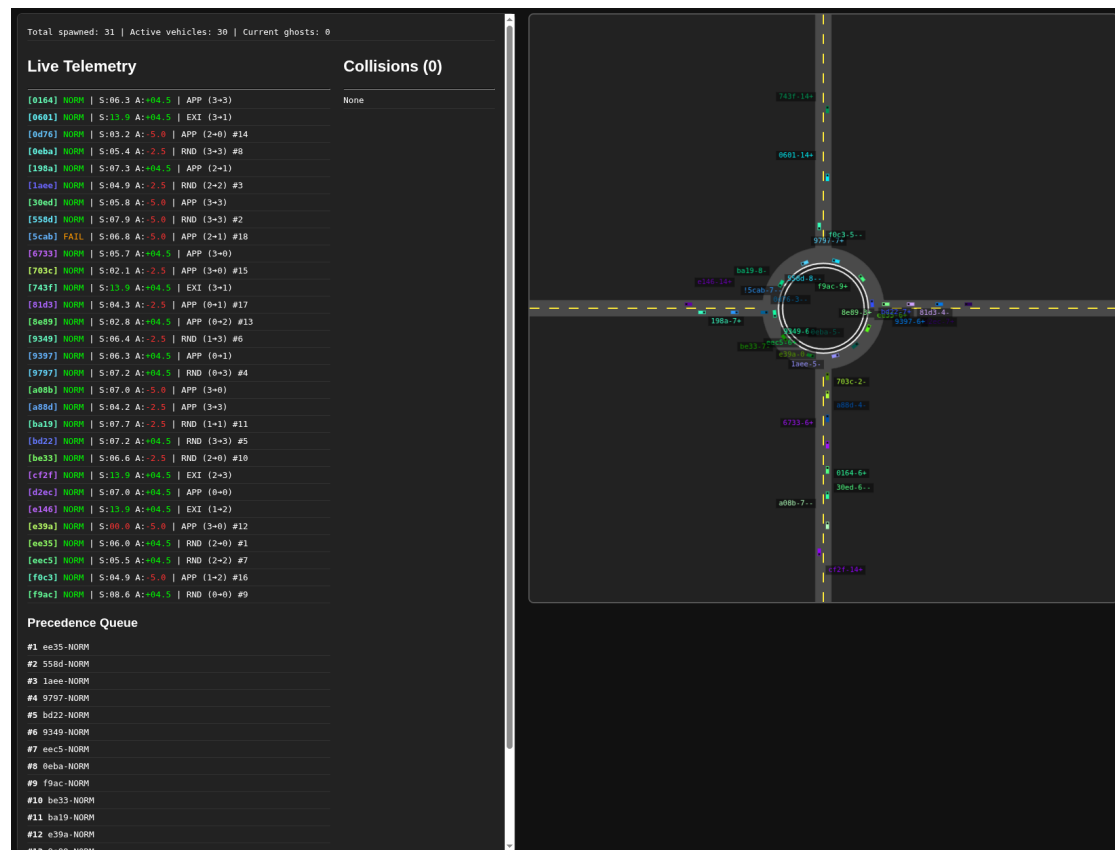


Figure 5: Screenshot of the running web page.

“A”; a shortened word for its navigation status (“APP” for approaching, “RND” for in-roundabout, “EXI” for exiting); start and destination road indexes (start number - j end number); position in the priority queue, if present in it;

- below the telemetry, the controller’s precedence queue, showing the vehicles with highest priority on the top;
- on the right, the identified collisions. Each one shows: a timestamp (*hours:minutes:seconds*), conflicting vehicles ($v_1 - v_2$), conflicting vehicles’ navigation statuses.

The actual **simulation** is shown in the area at the center of the page. Each vehicle is represented with a random color and a label in a random position around it; both the label and the id in the dashboard appear of the same colour. The car’s front is represented by its windshield.

Ghosts, as identified by the controller, also appear as grey cars. If a ghost is displayed over a real vehicle, it means that the controller hasn’t been able to merge them; most likely, this is the expected behaviour: it may mean that the vehicle is disconnected but the controller has successfully created a corresponding ghost in its place.

Pressing the *spacebar* will pause or resume the simulation.

The web server also exposes some http endpoints for interacting.

The bash script *src/ctrl.sh* can be used directly. It expects one command, optionally followed by the vehicles to address:

- without specified vehicles, the command is addressed to everyone (e.g. useful for pausing or resuming);
- if it’s a number N of one or two digits, N random vehicles will be addressed;
- otherwise, a string (different from the previously described N) will be used as prefix of a vehicle id and the single vehicle will be addressed, if found and unique.

Commands can be written as entire words or by their short letter, uppercase or lowercase:

- *pause* (p)
- *resume* (r)
- *failsafe* (f): forcefully set to failsafe
- *controller* (c): undo failsafe (set to normal)
- *disconnect* (d): forcefully set to disconnected
- *reconnect* (n): undo disconnect (set to normal, like for c)

Examples:

```
# Usage: src/ctrl.sh {PAUSE|RESUME|FAILSAFE|CONTROLLER|DISCONNECT|RECONNECT}
#           [vehicle-id|vehicle-count]
#   or src/ctrl.sh {p|r|f|c|d|n} [vehicle-id|vehicle-count]

# pause/resume simulation
./src/ctrl.sh p
./src/ctrl.sh r
./src/ctrl.sh resume
# enter failsafe
./src/ctrl.sh f
# exit failsafe, for vehicle with ID starting with 4fd2
./src/ctrl.sh c 4fd2
# mark 10 random vehicles as disconnected
./src/ctrl.sh d 10
# exit disconnected
./src/ctrl.sh n
```

These commands can be used to inject some chaos in the simulation and see how the vehicles behave; pausing also allows to analyze better the situation and, for example, to pick a precise vehicle, change its status and see if this will cause an accident.

The corresponding http API endpoints are:

- **GET /api/config:** retrieves the static geometrical configuration of the simulated roundabout. Returns a JSON object containing physical constants such as `r_radius`, `lane_width`, and `car_length`.
- **GET /api/state:** retrieves the current real-time state of the distributed system. Returns a JSON object containing the active `vehicles` dictionary, the `precedence_queue`, identified `ghosts` (disconnected nodes), and a list of detected collisions.
- **POST /api/control:** submits a `SystemCommand` to alter the simulation state or inject network faults. Accepts a JSON payload requiring a `command` string (*PAUSE*, *RESUME*, *ENTER_FAILSAFE*, *EXIT_FAILSAFE*, *ENTER_DISCONNECTED*, *EXIT_DISCONNECTED*). Optionally accepts `vehicle_id` to target a specific node, or `vehicle_count` to apply the fault to a random subset of nodes, otherwise will target all if left empty.

8 Self-evaluation

This project has proven itself an engaging testing ground to analyze some of the challenges of distributed systems in a simulated real-world, tangible scenario. Many architectural choices have been taken (section 3) and techniques have been studied and applied (behaviour section, 3.5) trying to address project's requirements and common

distributed systems problems, while other possible methods have been only discussed for future studies.

Observing the simulation, cars behave almost as expected: the traffic flows fluidly, especially with the use of the controller; cars yield, slow down or accelerate when they should and maintain the proper safety distance; vehicles in failsafe mode are more cautious; the controller can identify ghosts with sufficient precision when there aren't too many disconnected vehicles, so that it can be aware of them in its calculations. Even if a vehicle is abruptly set to disconnected, this still won't cause accidents.

However, the implementation has revealed itself trickier than expected. There are still bugs or imprecisions in managing the physics and the collisions. Therefore there are more collisions than we'd have hoped for, especially with many vehicles in failsafe mode, indicating that the failsafe logic may most of all require more improvements.

Some example results (from the testing section, 5.2), coming from averaging multiple 10 minutes simulation runs and roughly rounding their results to get an idea, using 4 roads, are: about 1 collision per minute with 40 vehicles in normal mode, 2 per minute with 20 vehicles, 0.3 with 10 vehicles; about 1 collision every 10 seconds with 40 vehicles, of which 10 to 30 are in failsafe or disconnected mode. Moreover, one run with 40 vehicles out of 40 in a disconnected state ended up in a deadlock inside the roundabout (same scenario described in 3.5.1).

The traffic flow as well doesn't seem to change much with or without the controller's help, since the total number of spawned vehicles is always around 300, for the same 40 vehicles test configurations as above.

The trickiest part is always to manage the collisions between a vehicle about to enter the roundabout and one inside. Of course, one could have set the entering vehicle to always stop and only enter if the roundabout is empty or almost empty, but that would have made the project not very interesting: the main point was exactly to try to make the vehicles coordinate with each other, while maintaining the safety and optimizing the overall traffic flow.

Given the available time for the development, correcting the collisions bugs would have meant neglecting many of the distributed systems aspects, so this way of managing the project has been chosen for its higher educational value and stimulating appeal. Regarding the controller's logic, then, surely a more efficient evaluation process would have made the traffic more efficient, but algorithms are yet another subject, not worth spending time on, in the context of a small distributed systems project: the implemented controller's checks were the sole ones sufficient to make the traffic flow and, then, to address the distributed systems problems.

Another weak point has of course been testing, widely discussed in 5.

9 Future works

As follows from the self-evaluation section (8), one of the most urgent steps would be to fix the logic implementation problems and make the simulation more correct (aiming at 0 collisions). Preventing the “roundabout deadlocks” (3.5.1), especially with many disconnected vehicles, would also be of high importance.

Then, as highlighted throughout the report, security hasn’t been considered at all: one could start from the suggestions in the design section (3.9).

Once these fundamentals have been taken into account and solidly validated, the development could proceed to add features, improve the controller’s logic to make it even more efficient and expand the simulation. Additional features may include more road or roundabout lanes, obstacles on the road, different vehicles or cars with different speed and acceleration parameters. For the controller, better algorithms could be implemented. Finally, the simulation could be expanded to a larger scale, for example integrating it into the context of more roads in a city, which could lead to new distributed systems problems.