DPS Project 2022

Distributed and Pervasive Systems Lab Course

April 2022

1 Project description

The goal of the project is to develop SETA (*SElf-driving TAxi service*), a peer-to-peer system of self-driving taxis for the citizens of a smart city.

Figure 1 shows the architecture of SETA. The smart city is divided into

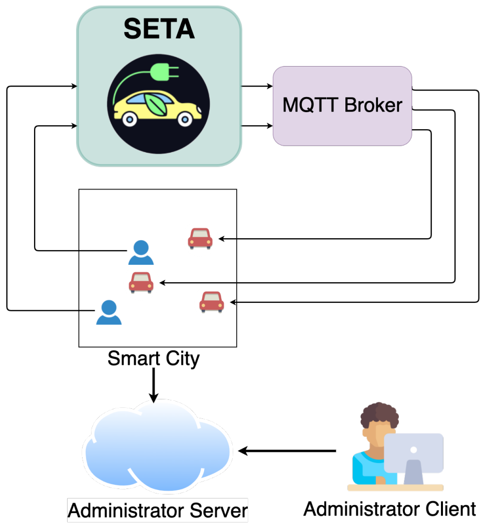

Figura 1: Overall architecture of SETA

four districts. The citizens of the smart city use SETA to request a self-driving taxi that takes them from their current position to a destination point. In SETA, the taxis of the smart city coordinate with each other to decide which taxi will handle each of such requests.

Each taxi is also equipped with a sensor that periodically detects the air pollution level of the smart city. Moreover, every ride consumes the battery level of the taxi. When the residual battery level is too low, the taxi must go to the recharge station of the district in which it is positioned. Periodically, the taxis have to communicate to a remote server called *Administrator Server* information about the air pollution levels, the number of completed rides, the number of kilometers driven, and the remaining battery level. The SETA administrators (*Administrator Client*) are in charge of monitoring this information through the *Administrator Server*. Furthermore, through the *Administrator Server* it is also possible to dynamically register and remove taxis from the system.

1.1 Smart city internal representation

The smart city is represented as a 10×10 grid (see Figure 2), divided into four 5×5 districts. Each cell of the grid represents a square kilometer of the

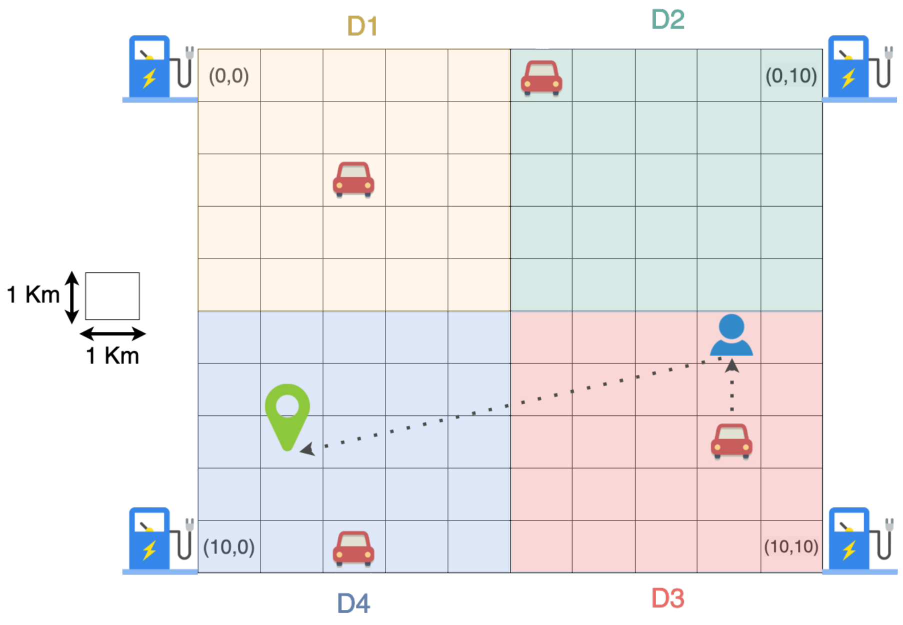

Figura 2: smart city representation

smart city. At a specific time, each cell may contain an arbitrary number of taxis (i.e., 0, 1, more than 1). When a taxi is registered within SETA, it will be randomly placed at the recharge station cell of one of the districts (i.e., $(0, 0)$, $(0, 10)$, $(10, 0)$, or $(10, 10)$).

2 Applications to be implemented

For this project, you are required to develop the following applications:

- *MQTT Broker*: the *MQTT* broker used to communicate the ride requests to the taxis
- *SETA*: a process that simulates the taxi service requests generated by the citizens that must be communicated to the fleet of self-driving taxis through *MQTT*

- *Taxi*: a specific self-driving taxi of the system
- *Administrator Server*: REST server that receives statistics from the smart city taxis and dynamically adds/removes taxis from the network
- *Administrator Client*: a client that queries the *Administrator Server* to obtain information about the statistics of the taxis and their rides

Please, note that each *Taxi* is a stand-alone **process** and, so, it **must not** be implemented as a thread.

In the following, we provide more details about the applications that must be developed.

3 MQTT Broker

The *MQTT Broker* on which SETA relies is online at the following address: `tcp://localhost:1883`.

SETA uses this broker to communicate the ride requests of the smart city citizens to the taxis. As it will be described in the next section, SETA uses a dedicated topic for the requests originating in each district of the smart city. SETA publishes on these topics, while the taxis subscribe to them in order to receive the ride requests.

4 SETA

SETA is a process that simulates the taxi service requests generated by the citizens of the smart city. Specifically, this process generates 2 new ride requests from the citizens every 5 seconds. Every ride request is characterized by:

- ID
- Starting Position
- Destination Position

Starting and destination points are expressed as the Cartesian coordinates of a smart city's grid cell. Such coordinates must be randomly generated. Each ride's starting and destination points may be located in the same district as well as in different districts.

To communicate the generated rides to the taxis of the smart city, SETA relies on the *MQTT Broker* presented in Section 3. Specifically, SETA assumes the role of publisher for the following four *MQTT* topics:

- `seta/smartcity/rides/district1`
- `seta/smartcity/rides/district2`

- `seta/smartcity/rides/district3`
- `seta/smartcity/rides/district4`

Whenever a new ride request with starting point included in the district i is generated, SETA publishes such a request on the following *MQTT* topic:

`seta/smartcity/rides/district{i}`

Note that, it is also possible to include additional *MQTT* topics if needed during the project development. SETA may also assume the role of subscriber for such additional topics.

5 Taxi

Each *Taxi* is simulated through a **process**, which is responsible for:

- coordinating itself with the other taxis by using **gRPC** to decide which taxi will take charge of each ride request generated by SETA
- accomplishing the rides it takes charge of: bring the citizen from the starting point to the destination point of the ride
- sending its local statistics (e.g., measured air pollution levels, traveled kilometers) to the *Server Administrator*

5.1 Initialization

A *Taxi* is initialized by specifying

- ID
- listening port for the communications with the other taxis
- *Administrator Server*'s address

Moreover, the initial battery level of each taxi is equal to 100%. Once it is launched, the *Taxi* process must register itself to the system through the *Administrator Server*. If its insertion is successful (i.e., there are no other taxis with the same ID), the *Taxi* receives from the *Administrator Server*:

- its starting position in the smart city (one of the four recharge stations distributed among the districts)
- the list of the other taxis already present in the smart city

Once the *Taxi* receives this information, it starts acquiring data from the pollution sensor. Then, if there are other taxis in the smart city, the taxi presents itself to the other taxis by sending them its position in the grid. Finally, the taxi subscribes to the *MQTT* topic of its district. Note that, it is also possible to include additional *MQTT* topics if needed during the project development. A taxi may assume both the role of publisher and subscriber for such additional topics.

As it will be explained in Section 5.2.1, the taxis of the smart city coordinate with each other to decide which taxi will handle each ride request. When a *Taxi* takes charge of a ride, it simulates the delivery time with a `Thread.sleep()` of 5 seconds. You can assume that taxis always successfully accomplish their rides. If the destination point of the ride is in another district compared to the one of the starting position, then the *Taxi* which accomplished the ride has to unsubscribe from the *MQTT* topic of the starting district. Then, it will subscribe to the *MQTT* topic of the destination district.

5.2 Distributed synchronization

5.2.1 Rides management

The taxis of each district must use a distributed and decentralized algorithm to decide who will take charge of each ride. Specifically, each ride will be handled by the *Taxi* (located in the same district of the ride's starting position) which meets the following criteria:

- the *Taxi* must not be already involved in another ride or a recharge process
- the *Taxi* must have the minimum distance from the starting point of the ride
- if more taxis meet the previous criteria, it must be chosen among them the *Taxi* with the highest battery level
- if more taxis meet the previous criteria, it must be chosen among them the *Taxi* with the highest ID

To compute the distance between two points $P_1 = (x_1, y_1)$ e $P_2 = (x_2, y_2)$ of the smart city, you should use the following formula:

$$d(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

If in a district there is no *Taxi* that is available to take charge of a ride, such a ride must not be discarded. Hence, you have to find a way to properly handle these rides.

Moreover, while the taxis decide who will take charge of a ride, there could be one or more taxis joining or leaving the smart city or changing the district in which they are positioned. Make sure to handle those cases during the development of the project.

Note that, all the communications between the taxis must be handled through *gRPC*.

5.2.2 Sending information to the server

Every 10 seconds, each *Taxi* has to compute and communicate to the administrator server the following local statistics:

- The number of kilometers traveled to accomplish the rides of the taxi
- The list of the pollution levels measurements
- The current battery level of the taxi

Once the local statistics have been computed, the *Taxi* sends them to the *Administrator Server* coupled with the timestamp in which they were computed.

5.3 Battery consumption

A *Taxi* consumes 1% of its battery level for each kilometer traveled to accomplish a ride. For simplicity, it is assumed that the battery consumption is computed only at the end of each ride.

When the battery level of the *Taxi* is below 30%, the taxi must go to the recharge station of its district. A recharge station may be accessed only by a single taxi at a time. It is also possible to explicitly ask a *Taxi* to recharge its battery through a specific command (i.e., *recharge*) on the command line. In both cases, you have to implement one of the *distributed* algorithms of *mutual exclusion* introduced in the theory lessons in order to coordinate the recharge operations of the taxis of each district. For the sake of simplicity, you can assume that *the clocks of the taxis are properly synchronized and that the timestamps of their requests will never be the same (like Lamport total order can ensure)*. If a taxi takes part to the mutual exclusion algorithm in order to recharge its battery, it cannot accept any rides until the recharging process of its battery is completed. Moreover, when a taxi acquires rights to recharge its battery:

- it consumes 1% of its battery level for each kilometer traveled to reach the recharge station
- its position becomes the same as the cell of the recharge station of the district in which the taxi is currently positioned.

The recharging operation is simulated through a `Thread.sleep()` of 10 seconds.

5.4 Explicit closure

It is assumed that each *Taxi* terminates only in a controlled way. Specifically, only when the message *"quit"* is inserted into the command line of a taxi process, the taxi will leave the system.

In both cases, to leave the system, a *Taxi* must follow the next steps:

- complete the possible ride it is involved in, sending to the *Administrator Server* the information described in Section 5.2.2
- complete any battery recharge
- notify the other taxis of the smart city
- request the *Administrator Server* to leave the smart city

5.5 Pollution sensors

Each taxi is equipped with a sensor that periodically detects the air pollution level of the smart city. Each pollution sensor periodically produces measurements of the level of fine particles in the air (PM10). Every single measurement is characterized by:

- PM10 value
- Timestamp of the measurement, expressed in milliseconds

The generation of such measurements is produced by a simulator. In order to simplify the project implementation, it is possible to download the code of the simulator directly from the page of the course on *Moodle*, under the section *Projects*. Each simulator assigns the number of seconds after midnight as the timestamp associated with a measurement. The code of the simulator must be added as a package to the project, and it **must not** be modified. During the initialization step, each *Taxi* launches the simulator thread that will generate the measurements for the air pollution sensor.

Each simulator is a thread that consists of an infinite loop that periodically generates (with a pre-defined frequency) the simulated measurements. Such measurements are added to a proper data structure. We only provide the interface (*Buffer*) of this data structure that exposes two methods:

- *void add*$(Measurement\ m)$
- *List* $<Measurement>$ *readAllAndClean*$()$

Thus, it is necessary to create a class that implements this interface. Note that each *Taxi* is equipped with a single sensor.

The simulation thread uses the method *addMeasurement* to fill the data structure. Instead, the method *readAllAndClean*, must be used to obtain the measurements stored in the data structure. At the end of a read operation, *readAllAndClean* makes room for new measurements in the buffer. Specifically, you must process sensor data through the *sliding window* technique that was introduced in the theory lessons. You must consider a buffer of 8 measurements, with an overlap factor of 50%. When the dimension of the buffer is equal to 8 measurements, you must compute the average of these 8 measurements. A *Taxi* will send these averages to the *Administrator Server* with the other information about the ride it accomplished (see Section 5.2.2 for more information).

6 Administrator server

The *Administrator Server* collects the IDs of the taxis registered to the system, and also receives from the taxis their local statistics (see Section 5.2.2). This information will then be queried by the administrators of the system (*Administrator Client*). Thus, this server has to provide different REST interfaces for:

- managing the taxis network
- receiving the local statistics from the taxis
- enabling the administrators to execute the queries

6.1 Taxis interface

6.1.1 Insertion

The server has to store the following information for each taxi joining the smart city:

- ID
- IP address
- The Port Number on which it is available to handle the communication with the other taxis

Moreover, the server is in charge of assigning to each joining taxi a randomly chosen district of the smart city. A *Taxi* can be added to the network only if there are no other taxis with the same identifier. If the insertion succeeds, the *Administrator Server* returns to the *Taxi*

- the starting position of the taxi in the smart city, i.e., the position of the recharge station of the randomly chosen district
- the list of taxis already located in the smart city, specifying for each of them the related ID, IP address, and the port number for communication

6.1.2 Removal

Whenever a *Taxi* asks the *Administrator Server* to leave the system, the server has to remove it from the data structure representing the smart city.

6.1.3 Statistics

The *Administrator Server* must provide an interface to receive the local statistics from the taxis of the smart city. These data have to be stored in proper data structures that will be used to perform subsequent analysis. During the development of the project, make sure that you correctly synchronize read and write operations made on these data structures. Indeed, the taxis of the smart city could send their local statistics while the Administrator Client is requesting to the Administrator Server to perform some computations on such statistics.

6.2 Administrator interface

When requested by the Administrator Client through the interface descripted in Section 7, the *Administrator Server* must be able to compute the following statistics:

- The list of the taxis currently located in the smart city
- The average of the last n local statistics of a given *Taxi*. In particular, it has to compute the average number of:
  - the travelled kilometres
  - the battery level
  - the pollution level
- The average of the previously introduced statistics provided by all the taxis and occurred from timestamps *t1* and *t2*

7 Administrator Client

The *Administrator Client* consists of a simple command-line interface that enables interacting with the *REST* interface provided by the *Administrator Server*. Hence, this application prints a straightforward menu to select one of the services offered by the administrator server described in Section 6.2 (e.g., the list of the smart city taxis), and to enter possible required parameters.

8 Simplifications and restrictions

It is important to recall that the scope of this project is to prove the ability of designing and building distributed and pervasive applications. Therefore, all the aspects that are not strictly related to the communication protocol, concurrency, and sensory data management are secondary. Moreover, it is possible to assume that:

- no nodes terminate in an uncontrolled way
- no nodes behave maliciously

On the contrary, you should handle possible errors in data entered by the user. Furthermore, the code must be robust: all possible exceptions must be handled correctly.

Although the Java libraries provide multiple classes for handling concurrency situations, for educational purposes it is mandatory to **use only the methods and classes explained during the laboratory course.** Therefore, any necessary synchronization data structures (such as lock, semaphores, or shared buffers) should be implemented from scratch and will be discussed during the project presentation. Considering the communication between the taxis' processes, it is necessary to use the *gRPC* framework. If broadcast communications are required among taxis, these must be carried out in parallel and not sequentially.

9 Project presentation

You must develop the project individually. During the evaluation of the project, we will ask you to discuss some parts of the source code and we will also check if it runs correctly considering some tests. Moreover, we will ask you one or more theoretical questions about the theoretical content of the lab lessons. You will run your code on your machine, while the source code will be discussed on the tutor's machine. You must submit the source code before discussing the project. For the submission, you should archive your code in a *.zip* file, renamed with your university code (i.e., the "matricola"). For instance, if your university code is 760936, the file should be named 760936.zip. Then, the zip file should be uploaded at *http://upload.di.unimi.it.* It will be possible to submit your project a week before the exam date. You must complete the submission (strictly!) by two days before the exam date at 11:59 PM (e.g., if the exam is on the morning of the 15th, you must complete the delivery before the 11:59 PM of the 13th).

We suggest the students to use during their project discussion the `Thread.sleep()` instructions to show how the synchronization problems have been correctly handled. We recommend you to analyze all the synchronization issues and to create schemes to illustrate how the components of your project communicate. These schemes may represent the format of the messages and the sequence of the communications that occur among the components during the different operations involved in the project. Such schemes can be very helpful during the project discussion. It is not necessary to show the structure of the classes of your project, but only the main aspects regarding the synchronization and the communication protocols you have implemented.

During the presentation of the project, we will ask you to test your code by launching at least 4 or 5 taxis and one Administrator client.

10 Plagiarism

The reuse of code written by other students will not be tolerated. In such a case, we will apply the sanctions described course website, in the section *Student Plagiarism* of *General Information.*

The code developed for the project **must not** be published (e.g., github) at least until March of the next year.

11 Optional parts

In order to encourage the presentation of the project in the first exams sessions, the development of the first optional part becomes mandatory from the September's session (included), while both the first and the second optional parts are mandatory from the January's session (included).

11.1 First part

In a distributed system, it is very likely that the clocks of the different devices are not synchronized. For this optional part, the assumption that the clocks are synchronized is no longer valid. Thus, you must implement the *Lamport* algorithm to solve this problem. In particular, considering the recharging operation, you have to use the *Ricart and Agrawala* algorithm to coordinate the taxis in a distributed fashion. The timestamps of the messages that the taxis use to implement the *Ricart and Agrawala* algorithm cannot be generated only through the method `System.currentTimeMillis()`. During the initialization steps, each *Taxi* must also generate a random offset that is added to every timestamp to simulate the mismatch between the clocks of the different taxis.

11.2 Second part

We previously assumed that the processes can terminate only in a controlled way. This assumption is no longer valid for the taxis in this optional part. Hence, all the distributed synchronization mechanisms implemented for the project must take into account that a *Taxi* could leave the system in an uncontrolled way.

12 Updates

If needed, the current text of the project will be updated. Changes in the text will be highlighted. Please, regularly check if new versions of the project have been published on *Moodle*.