Smart Parking

Internet of Things - Final Project

A.Y. 2016-2017

Rosa Mirco

Student ID: 279420

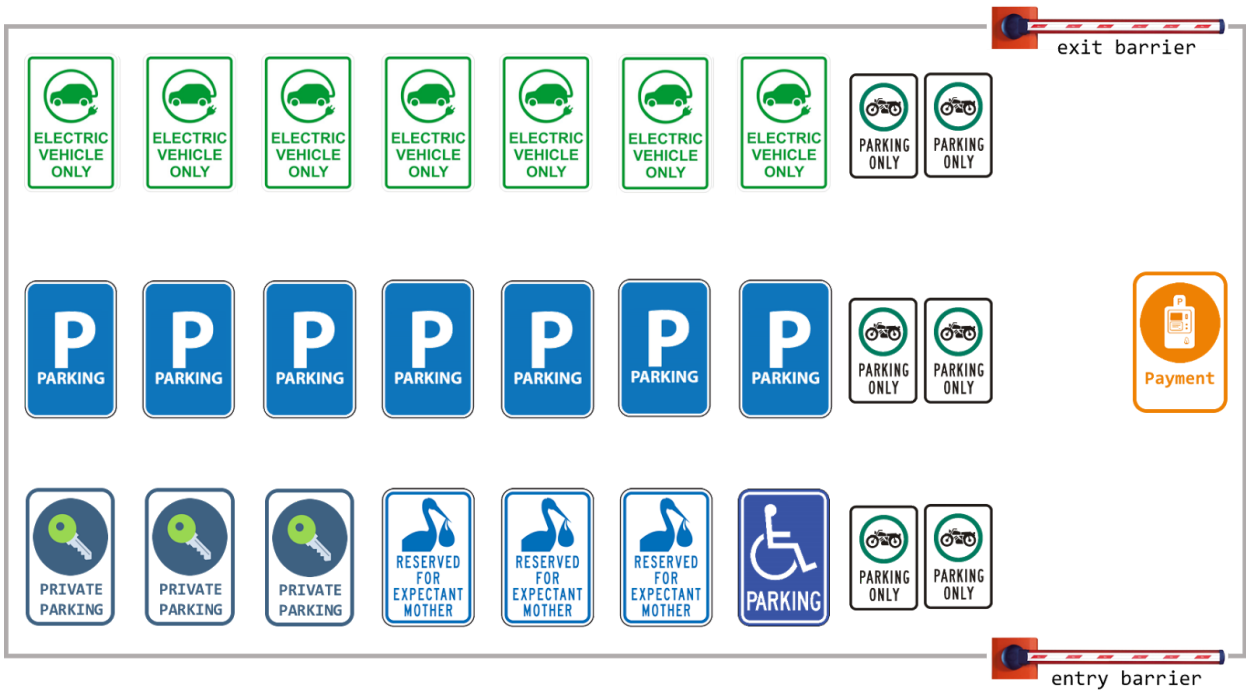
Mail Address: mirco.rosa@studenti.unipr.it

Table of Contents

Overview

The main objective of this project is to build a complete data exchange infrastructure for a parking lot like the one represented in the picture below. It will have to manage all the activities related to a scenario like this, such as:

- Access control to the structure (entry/exit barriers)
- Ticket managing
- Granular access control to parking lots
- Payments (calculation, payment methods, etc.)
- Secure access to private parking lots
- Backup for electrical parking events



All this functionalities will be implemented using Californium, a Java implementation of CoAP, in order to take advantages of the aforementioned protocol features. Before moving on to the detailed description of all the components, it is useful to have a look to some basic objects that will be used by both control devices and vehicles that will use the parking lot.

Basic shared objects

As said before, the objects described in this section are common to both control devices and vehicles. Also, we will not use the `ParkingStorage` object (as proposed in the outline) implementing a more realistic storage managed by the control devices and accessed through proper CoAP mechanisms.

Ticket

This is one of the two pivotal objects that the entire infrastructure will store and use extensively. It is a JavaBean containing all the fields necessary to support the operations along the way, such as:

- Ticket code
- Parking lot name
- Plate (only for electrical vehicles)
- Payment method (stored later)
- Release date
- Parking date
- Leaving date
- Payment date

It's important to note that date fields are stored as Strings and not as Date object, allowing the use of Json as data exchange format. Moreover, it has been chosen a particular pattern for the ticket code:

```
type code(3 letters) + incremental number for the specific type (5 numbers)
```

This pattern is optional as the system will work with any format we want, but is particularly useful for logs readability.

Parking Lot Info

This is the other main object used. As the previous one, is a Javabean used to describe the status of a single parking lot that uses data types suitable for Json-based communication:

- Name
- Type
- TIcket code
- Plate (only for electrical vehicles)
- Level
- Number
- Occupied (true/false)

As for ticket code it has been used a pattern for parking lot name:

`lottype_floor-number`

In this case the name of the parking lot is a structural choice and is used by the components to obtain informations, so it cannot be changed.

Preferences and Types

In order to allow flexible editing of parameters for both parking and simulation elements, the `Prefs` class is extensively used: it contains static variables such as parking generation sizes and ratios, port numbers, probabilities and delay times for the simulator, etc. On the other hand, types of lots, payments and vehicles are defined in `enum` classes ensuring flexibility on changes.

Smart Parking

This section will explain in detail all the elements composing the smart parking, and how scenario is generated at the beginning. The focus here is on the single components, while the interaction between them is discussed in the next chapters.

Components and behaviours

Entry Barrier

The entry barrier, like the other control devices, is a CoAP server that exposes several resources:

- **TicketEmitter**: emits a new ticket in response to a POST request that includes the vehicle type in the payload (also the plate if electric). If all is ok is provided a Json containing ticket informations, otherwise is returned a CoAP response code depending on the exception occurred. In case of GET requests the total number of emitted tickets is returned;
- **TicketTypeCounter**: one for each vehicle type, returns the total number of tickets emitted for that particular type in response to a GET request.

- **EntryBCoapClient:** this CoAP client is used to send informations related to the newly emitted ticket to the Payment Box, which is responsible for storing data properly (as we will see lately, Private and Hand tickets are treated as already paid).

Here is the POST method of the `TicketEmitter` class:

```
@Override
public void handlePOST(CoapExchange exchange) {
 //Pattern for Electric vehicle: type:plate
 String[] payload = exchange.getRequestText().split(":");
 DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy HH:mm:ss");
 Date currentTime = Calendar.getInstance().getTime();
 LOG.info(payload[0]);
 if(typeIsOk(payload[0])) { //Checks for unknown types
  server.incrementTypeCounter(payload[0]);
  StringBuilder ticket_code = new StringBuilder()
     .append(payload[0].substring(0,3).toUpperCase()) //Clients send Enum string as identifier
     .append(String.format("%05d",server.getTypeCount(payload[0])));
  LOG.info("New ticket emitted: "+ticket_code.toString());
  //If is Electric, add plate number to the ticket
  String json = "";
  if (payload.length==1)
   json = new Gson().toJson(new Ticket(ticket_code.toString(),dateFormat.format(currentTime)));
  else if (payload[0].equals(VehicleType.ELECTRIC.getVehicleName()) && payload.length==2)
   json = new Gson().toJson(new
Ticket(ticket_code.toString(),dateFormat.format(currentTime),payload[1]));
  else {
   LOG.severe("Error in ticket request");
   exchange.respond(CoAP.ResponseCode.NOT_ACCEPTABLE);
  }

  if(!payload[0].equals(VehicleType.PRIVATE.getVehicleName()) &&
payload[0].equals(VehicleType.HAND.getVehicleName()))   //Not private or hand
   server.putStandardTicketToPaymentBox(json);
  else  //Private and Hand ticket, managed as already paid
   server.putPrivateHandTicketToPaymentBox(json);
  exchange.respond(CoAP.ResponseCode.CONTENT,json,MediaTypeRegistry.APPLICATION_JSON);
  LOG.info("Json: "+json);
 }
 else {
  LOG.info("Lot type not recognized");
  exchange.respond(CoAP.ResponseCode.NOT_ACCEPTABLE);
 }
}
```

Payment Box

This CoAP server is the heart of the entire infrastructure, as it manages most of the operations (as we will see in detail later in the *Operations and mechanisms* part); for now, we only focus on its components:

- **PaidLots** and **UnpaidLots**: those resources deal with all the activities related to ticket lists, such as addition, removal and queries; PUT requests trigger updates of the interested list, while a GET request will return the entire ticket list in Json format;
- **ParkingPayment**: this resource, as the name says, is responsible for payments. It handles POST request accepting as input from payload the ticket code and the payment method and returning, after the necessary processing, the ticket updated with dwell time and total cost. Here we can see the payment process:

```
@Override
public void handlePOST(CoapExchange exchange) {
 //Payload format: ticketCode:paymentMethod
 String[] splitPayload = exchange.getRequestText().split(":");
 Ticket ticket = server.getUnpaidTicketByCode(splitPayload[0]);
 if(ticket!=null && isPaymentMethodValid(splitPayload[1])) { //Process Payment
  if(!ticket.getLeavingDate().isEmpty()) {
   DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy HH:mm:ss");
   Date currentTime = Calendar.getInstance().getTime();
   ticket.setPaymentDate(dateFormat.format(currentTime));
   ticket.setPaymentMethod(splitPayload[1]);
   payAndUpdateTicket(ticket);

   exchange.respond(CoAP.ResponseCode.CONTENT,new
Gson().toJson(ticket),MediaTypeRegistry.APPLICATION_JSON);
   LOG.info(new Gson().toJson(ticket));
   server.addPaidTicket(server.getUnpaidTicketByCode(splitPayload[0]));
   server.removeUnpaidTicket(splitPayload[0]);
  } else {
   LOG.warning("Not ready to pay");
   exchange.respond(CoAP.ResponseCode.FORBIDDEN);
  }
 } else if (server.getPaidTicketByCode(splitPayload[0])!=null) {
  LOG.warning("Ticket already paid.");
  exchange.respond(CoAP.ResponseCode.NOT_ACCEPTABLE);
 } else {
  LOG.warning("Ticket not found.");
  exchange.respond(CoAP.ResponseCode.NOT_FOUND);
 }
}
```

```
private void payAndUpdateTicket(Ticket ticket) {
 try {
  //For testing purposes, prices are considered and calculated €/second (to avoid waits)
  DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy HH:mm:ss");
  Date parkDate = dateFormat.parse(ticket.getParkingDate());
  Date leavingDate = dateFormat.parse(ticket.getLeavingDate());

  int parkingTimeSec = (int)((leavingDate.getTime() - parkDate.getTime()) / 1000 % 60);
  ticket.setDwellTime(parkingTimeSec);
  ticket.setTotalCost(parkingTimeSec*prices.get(ticket.getParkingLot().split("_")[0]));
 } catch (ParseException e) {
  e.printStackTrace();
 }
}
```

- **TicketValidityCheck:** the role of this resource is, at the request of other control devices, to query ticket lists (using `PaidLots` and `UnpaidLots` resources) and check if a specific ticket is valid for this or that particular operation. Here is one of the methods that performs the check:

```
@Override
public void handleGET(CoapExchange exchange) {
 LOG.info("Ticket to check: "+exchange.getRequestOptions().getUriQueryString());
 Ticket ticket =
server.getUnpaidTicketByCode(exchange.getRequestOptions().getUriQueryString());
 if(ticket!=null) {
  if(ticket.getParkingDate().isEmpty() || ticket.getLeavingDate().isEmpty())  //Valid ticket
   exchange.respond(CoAP.ResponseCode.VALID);
  else
   exchange.respond(CoAP.ResponseCode.FORBIDDEN);
 } else   //Unknown ticket
  exchange.respond(CoAP.ResponseCode.NOT_FOUND);
}
```

- **PBCoAPClient**: this element is extremely important, as it takes care of synchronizing the status of all the parking lots. At the systems startup an observing relation is created with each single parking lot in order to get and manage (using `CoapHandlers`) real-time updates every time there is a state change (parking/leaving); newly created relations are then bind with a `PLInfo` object (in a `ParkingLotRecord` object) and stored into a list, in order to provide a "fresh image" of the whole parking lot to whoever needs it. It's also important to say that `CoapHandlers` are `synchronized` in order to maintain data consistency while simultaneous calls are made to the handlers. Here is reported the `onload()` method:

```
@Override
public synchronized void onLoad(CoapResponse response) {
 //Updates received are right (errors handled directly by parking lot)
 PLInfo info = new Gson().fromJson(response.getResponseText(),PLInfo.class);

 if(isAnUpdate(info)) {  //To avoid 60-second refreshes of observing
   plInfo.setName(info.getName());
   plInfo.setType(info.getType());
   plInfo.setLevel(info.getLevel());
   plInfo.setNumber(info.getNumber());
   plInfo.setTicketCode(info.getTicketCode());
   plInfo.setPlate(info.getPlate());
   LOG.info("Received update: "+plInfo.getName()+" "+plInfo.getTicketCode());

   //Updating ticket time
   if(!plInfo.getTicketCode().isEmpty()) {   //Someone Parked
     Ticket ticketParkUpdate = (!plInfo.getType().equals(LotType.PRIVATE.getLotName()) &&
!plInfo.getType().equals(LotType.HAND.getLotName())) ?
server.getUnpaidTicketByCode(plInfo.getTicketCode()) :
server.getPaidTicketByCode(plInfo.getTicketCode());
     if(ticketParkUpdate!=null && ticketParkUpdate.getParkingDate().isEmpty()) {
       DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy HH:mm:ss");
       Date currentTime = Calendar.getInstance().getTime();
       ticketParkUpdate.setParkingDate(dateFormat.format(currentTime));
       ticketParkUpdate.setParkingLot(plInfo.getName());
       LOG.info(new Gson().toJson(ticketParkUpdate));
     } else
       LOG.info("You already parked");
   } else { //Someone leaved
     Ticket ticketLeaveUpdate = (!plInfo.getType().equals(LotType.PRIVATE.getLotName()) &&
!plInfo.getType().equals(LotType.HAND.getLotName())) ?
server.getUnpaidTicketByLotName(plInfo.getName()) :
server.getPaidTicketByLotName(plInfo.getName());
     if(ticketLeaveUpdate!=null && ticketLeaveUpdate.getLeavingDate().isEmpty()) {
       DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy HH:mm:ss");
       Date currentTime = Calendar.getInstance().getTime();
       ticketLeaveUpdate.setLeavingDate(dateFormat.format(currentTime));
       LOG.info(new Gson().toJson(ticketLeaveUpdate));
     } else if (ticketLeaveUpdate!=null)
       LOG.info("You already leaved the parking lot");
   }

 } else LOG.info("60 second refresh message");
}
```

Exit Barrier

The main task of the exit barrier is to decide if a vehicle is allowed to leave the parking lot. It is composed of:

- **DismissTicket**: this resource accepts two types of request: POST if the vehicle wants to exit, and GET to obtain the total number of dismissed tickets. To determine if that specific vehicle is allowed to exit the parking lot, the ticket code is sent to the Payment Box using the CoAP client described below.
- **ExitBCoapClient**: this CoAP client has the only function of checking if the ticket received has already been paid, sending a PUT request to the Payment Box and returning a response based on the response code obtained.

Parking Lots

`ParkingLots` are CoAP Servers whose function is to manage granular access to individual parking lots depending on type. Each server is typically composed by two elements: a `PLInfoRes` that handles request from vehicles, and a `PLCoapClient` used to communicate with the Payment Box. As said previously there are many types of parking lots, so here we have used the OOP model creating an abstract superclass `ParkingLot` and subsequently a subclass for each specific type.

Normal, Moto, Expectant

Those three types have a very similar behaviour, so we can group them together. As anticipated before, we have:

- **PLStandardInfoRes**: this resource can be reached by vehicles that want to enter the parking lot through a POST request with the payload format `1:ticketCode` (1 to park, 0 to leave). The response depends on the result obtained by both internal elaboration based on the current state of the parking lot and a "remote check" of the ticket on the Payment Box performed by the CoAP client.
- **PLStandardCoapClient:** this client is in charge of making the check request to Payment Box, returning a result depending on the answer received. This result is then passed to the aforementioned resource and processed with the other data.

Hand

The Hand type is similar to Normal, Moto and Expectant, with the difference that for Hand vehicles the parking is free. So no checks to Payment Box are made during the POST processing (ticket is considered already paid), and a CoAP client is not needed.

Private

Like Hand, there is no need of a CoAP client as the parking lot is free for the owner. The main difference here is that we use a secret code to grant access only to the owner and nobody else: to do that, in the POST payload is passed also the secret code with the format `1:ticketCode:secretCode`, and a check is performed with the code stored into the parking lot object. For testing purposes the code is the result of an SHA-1 hashing of the string `name+" Secret Code"`, in order to provide obfuscation for the data transmitted and a fixed length regardless the size of the plain text passcode. Here is the code generation and the POST method of the resource:

```
//For testing purposes, secret code is the hash of: lotName+" Secret Code"
try {
 MessageDigest messageDigest = MessageDigest.getInstance("SHA-1");
 String plainTextCode = name+" Secret Code";
 messageDigest.update(plainTextCode.getBytes());
 secretCode= new String(messageDigest.digest());
 LOG.info("Secret code: "+secretCode);
} catch (NoSuchAlgorithmException e) {
 e.printStackTrace();
}
...
@Override
public void handlePOST(CoapExchange exchange) {
 //Message format example: 1:ticketCode:secretCode
 String[] payload = exchange.getRequestText().split(":");

 if(payload.length==3) {
  if(payload[2].equals(secretCode)) { //Code confirmed
   LOG.info("Code confirmed - access granted");
   if((Integer.valueOf(payload[0])==1 && plInfo.isOccupied()) || (Integer.valueOf(payload[0])==0
&& !plInfo.isOccupied()))
    exchange.respond(CoAP.ResponseCode.BAD_REQUEST);
   else {
    if(!plInfo.isOccupied()) {
     plInfo.occupyParkingLot(payload[1]);
     changed();
     exchange.respond(CoAP.ResponseCode.VALID);   //Parked
    } else {
     if (plInfo.getTicketCode().equals(payload[1])) {
```

```
          plInfo.freeParkingLot();
          changed();
          exchange.respond(CoAP.ResponseCode.VALID);  //Parking lot freed
        } else {
          exchange.respond(CoAP.ResponseCode.FORBIDDEN);   //Parking lot occupied
        }
      }
    }
  } else {
    LOG.warning("Wrong code - not allowed to park");
    exchange.respond(CoAP.ResponseCode.UNAUTHORIZED);
  }
 } else
   exchange.respond(CoAP.ResponseCode.NOT_FOUND);   //No ticket code or secret code
}
```

Electric

The Electrical parking lot behaviour is similar to Normal, Moto and Expectant, with some main substantial exceptions: the plate of the vehicle is required, so the POST payload must follow the pattern `1:ticketCode:plate` in order to obtain the data needed for the next step; then a POST request is sent by the CoAP Client to the `ElectricalEventServer` (described in the next section), which handles the backup of parking and leaving events for electrical vehicles. Let's se the POST method in this case:

```
@Override
public void handlePOST(CoapExchange exchange) {
 //Message format example: 1:ticketCode:plate
 String[] payload = exchange.getRequestText().split(":");
 if(payload.length==3) {
   if((Integer.valueOf(payload[0])==1 && plInfo.isOccupied()) || (Integer.valueOf(payload[0])==0 &&
!plInfo.isOccupied())) {
     exchange.respond(CoAP.ResponseCode.BAD_REQUEST);
   } else if(coapClient.checkTicketValidity(payload[1])) {   //Checks from PB's ticket list
     if(!plInfo.isOccupied()) {
       plInfo.occupyParkingLot(payload[1]);
       plInfo.setPlate(payload[2]);
       changed();
       coapClient.postElectricalEvent(exchange.getRequestText());
       exchange.respond(CoAP.ResponseCode.VALID);   //Parked
     } else {
       if(plInfo.getTicketCode().equals(payload[1])) {
         plInfo.freeParkingLot();
         plInfo.emptyPlate();
         changed();
         coapClient.postElectricalEvent(exchange.getRequestText());
         exchange.respond(CoAP.ResponseCode.VALID);  //Parking lot freed
       } else {
```

```
        exchange.respond(CoAP.ResponseCode.FORBIDDEN);   //Parking lot occupied
       }
      }
     } else
      exchange.respond(CoAP.ResponseCode.NOT_ACCEPTABLE);
   } else
     exchange.respond(CoAP.ResponseCode.NOT_FOUND);   //No ticket code or plate
  }
```

Electrical Events Server

As mentioned above, the `ElectricalEventServer` has the task of manage and update the backup of parking and leaving events for electrical vehicles. All the data is stored in a list of `ElectricalEvent` JavaBean objects structured like this:

- Ticket code
- Plate
- Parking Date
- Leaving Date

The management of the list is performed by a particular resource:

- **AddElectricalEvent**: deals with POST request coming from Electrical parking lots, updating `ElectricalEvents` adequately and responding to clients with proper response codes. Furthermore, making a GET request is possible to obtain the full list of events in Json. Here is the code:

```
@Override
public void handlePOST(CoapExchange exchange) {
 //Message format example: 1:ticketCode:plate
 String[] payload = exchange.getRequestText().split(":");
 DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy HH:mm:ss");
 Date currentTime = Calendar.getInstance().getTime();
 if(payload[0].equals("1")) {  //Parking event
   events.add(new ElectricalEvent(payload[1],payload[2],dateFormat.format(currentTime)));
   LOG.info("Electrical event added.");
   exchange.respond(CoAP.ResponseCode.VALID);
 } else {  //Unparking event
   ElectricalEvent event = findParkingEvent(payload[1],payload[2]);
   if(event!=null) {
     event.setLeavingDate(dateFormat.format(currentTime));
     LOG.info("Electrical event completed");
     exchange.respond(CoAP.ResponseCode.VALID);
   } else {
     LOG.warning("Parking event not found");
     exchange.respond(CoAP.ResponseCode.NOT_FOUND);
   }
 }
}
```

Scenario generation and setup

It's time to put all the components described above together: the `SmartParking` class is the software entry point, and has the task of building the smart parking scenario accordingly with the parameters specified in the `Prefs` class. First of all it creates all the parking lots (using the `ParkingLotFactory` class) considering parameters like number of levels, ratios for each type, and level size factor; after this, the four control devices are initialized and, specifically for the Payment Box, all the observing relations with parking lot servers are setted up. After this process the system is online and ready to accept vehicles, ensuring robustness (unauthorized actions are managed properly) and short response times. Let's take a look at the methods used:

```
private void initializeSystems() {
 LOG.info("Initializing Access and Payment systems...");
 entryBarrier = new EntryBarrier(Prefs.ENTRY_BARRIER_PORT);
 paymentBox = new PaymentBox(Prefs.PAYMENT_BOX_PORT);
 exitBarrier = new ExitBarrier(Prefs.EXIT_BARRIER_PORT);
 electricalEventsServer = new ElectricalEventsServer(Prefs.ELECTRICAL_EVENTS_SERVER_PORT);
}

private void initializeParkingLots() {
 for (int levelNumber = 0; levelNumber < Prefs.LEVELS; levelNumber++) {
   generateLevel(levelNumber);
 }
}

//Lot identification name: type_level-number
private void generateLevel(int levelNumber) {
 for (LotType type : LotType.values()) {
   for (int i = 0; i < Prefs.PL_RATIOS[type.ordinal()] * Prefs.LEVEL_SIZE_FACTOR; i++) {
parkingLots.add(lotFactory.createParkingLot(type,levelNumber,i,Prefs.PL_STARTING_PORT+
parkingLots.size()));
   }
 }
}
```

Use Case Simulator

In order to test thoroughly the infrastructure described in the previous chapter, we have to set up a benchmark platform able to simulate a case of real use. For this purpose we have designed specific objects for each type of vehicle that act like potential real users; those objects are then instantiated as threads by the simulator, following particular criteria that we will describe in detail later.

Vehicles

Vehicles are modelled as CoAP clients that perform a series of requests to the control devices in the smart parking. Since many of the behaviours that we will use are common to several vehicles, again adopting the OOP model is useful: the abstract class Vehicle includes those general methods, that will be overridden properly in case of particular requirements:

- **enterParkingLot()**: this method makes a POST request to the Entry Barrier, getting back the emitted ticket if is allowed to enter or an error code otherwise;
- **lotSearch()**: after entering, the vehicle performs a random search for an empty parking lot that matches his type: iteratively (with a delay to make the search process more realistic) a random port is selected from the range of parking lot servers, and a GET is made to retrieve infos and determine if we can park. In case of a positive match, parking lot infos and port are stored in the vehicle, that is now ready to park. Here is the code:

```
//Works for all parking lots except Private and Electric
public void lotSearch() throws InterruptedException {
 int portRange = calculatePortRange();
 int port;
 PLInfo tryPLInfo;

 do {
  Thread.sleep(Prefs.PL_SEARCH_BASE_MILLS +sr.nextInt(Prefs.PL_SEARCH_RANGE_MILLS));
  port = Prefs.PL_STARTING_PORT+sr.nextInt(portRange);
  setURI("127.0.0.1:"+port+"/parking_lot");
  Request request = Request.newGet();
  tryPLInfo = new Gson().fromJson(advanced(request).getResponseText(),PLInfo.class);
  LOG.info("Trying to park at "+tryPLInfo.getName());
```

```
} while(!tryPLInfo.getType().equals(type.getVehicleName()) || !tryPLInfo.getTicketCode().isEmpty()
|| tryPLInfo.isOccupied());

plInfo=tryPLInfo;
parkingLotPort=port;
}
```

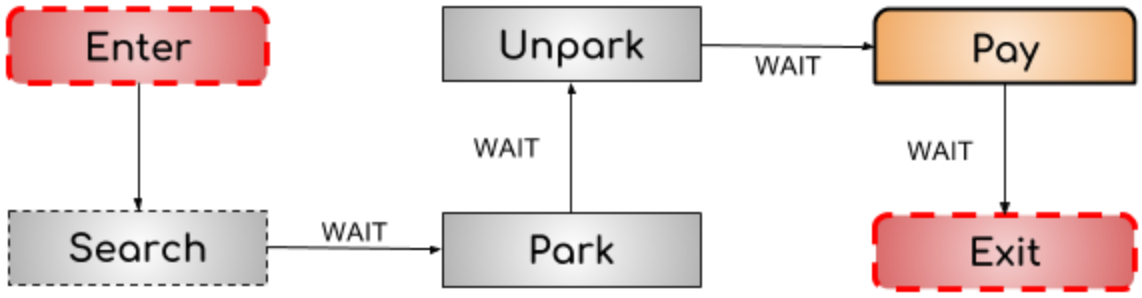
- **park()**: after obtaining a positive match in `lotSearch()`, a POST request is performed to the parking lot resource involved in managing the parking process, passing in the payload the integer 1 (which indicates our intention to park) and the ticket code: again, the success or failure of the operation will be determined by the response code got back;
- **unpark()**: this method is used to leave the parking lot: again in the POST request is passed the ticket code, but this time the integer is set at 0 (which indicates our intention to leave). As usual, various response codes are managed to handle all the different cases.
- **patTicket()**: as the name says, calling this method equals to perform a POST to the resource of Payment Box in charge for ticket payment. This time is passed the ticket code together with the payment method, and as response we will receive a Json representation of the updated ticket or an error code indicating which exception occurred.
- **exitParkingLot()**: this method makes a POST request to the exit barrier, providing a response code as response that tells us if we can leave the structure or if the ticket is still to be paid.

To allow a more sophisticated control on problems that can occur during the simulation, all the aforementioned methods are capable of throwing an exception called `VehicleException`, expressly designed to stop the execution in case of unwanted behaviours and provide a meaningful error message.

The following sections will explain individually the different types of vehicle that we are going to simulate. Note that between the operations have been added several sleep() calls, which duration is computed adding to the base value a random one (as shown in the following code), in order to provide a more realistic simulation of the operating times.

Normal, Moto, Expectant

These three categories have been grouped as their command sequence (shown in figure) is identical:



As we can see, those vehicles employ all the methods described before, so no override mechanisms are required. Here is the code:

```
@Override
public void run() {
 //Exceptions are used to stop the flow if some illegal action is performed (eg: 2 consecutive calls to park() or unpark())
 try {
   enterParkingLot();
   lotSearch();
   park();
   Thread.sleep(Prefs.PARKING_TIME_BASE_MILLS+sr.nextInt(Prefs.PARKING_TIME_RANGE_MILLS));
   unpark();
   Thread.sleep(Prefs.FROM_PL_TO_PB_BASE_MILLS+sr.nextInt(Prefs.FROM_PL_TO_PB_RANGE_MILLS));
   payTicket();
   Thread.sleep(Prefs.EXIT_BASE_MILLS +sr.nextInt(Prefs.EXIT_RANGE_MILLS));
   exitParkingLot();
 } catch (VehicleException e) {
   LOG.severe("Something goes wrong during operations: "+e.getMessage());
 } catch (InterruptedException e) {
   e.printStackTrace();
 }
}
```

Hand

Hand vehicle behaviour is almost the same of the one described above, with the obvious exception of payment as for this category parking is free; however if a payment action is performed the system will notify the client properly and the user will not pay anything.

Private

In case of Private vehicles, things are a bit different and some methods are overridden:

- **lotSearch()**: as the private client can park only in his parking lot, the random search has been substituted with a sequential one, that parses the parking lot port range until it finds the right one;
- **park()** and **unpark()**: in addition to integer identifier and ticket code, the secret code is passed in order to park. For testing purposes the secret code is generated in the same manner as the parking lot does, but the mechanism works with any other methods;

Again in this case no payment is due, so there is no need to call the corresponding method (even if the Payment Box handles this occurrence in the same way as Hand vehicles).

Electrical

Electrical vehicles need to perform some additional operations:

- **enterParkingLot()**: as described in the previous chapter, the POST request to the Entry Barrier also includes the plate, in order to store the parking event to the Electrical Event Server correctly;
- **park()** and **unpark()**: again, in addition to integer identifier and ticket code we have to pass the plate to the parking lot, allowing the correct submission of the information to the dedicated server.

Simulation mechanisms and options

Last but not least, the simulation environment. The `UseCaseSimulator` class is the entry point for the simulator, and takes care of launching and monitoring the submission of vehicles during the process. The first operation is the creation of the probability array, based on specific ratio parameters for each type of vehicle:

```
private void generateTypeProbabilityArray() {
 int probs_size = 0;
 for(int type_prob : Prefs.VEHICLES_PROBABILITY_RATIOS)
   probs_size+=type_prob;
 probs = new int[probs_size];
```

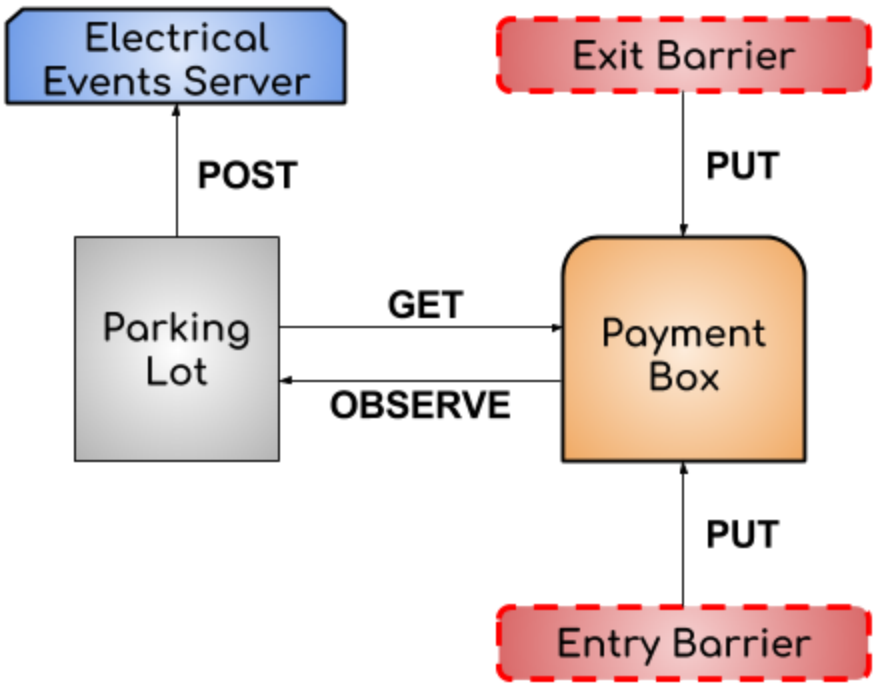
```
 //Reusing probs_size as index
 for (int i = 0; i < Prefs.VEHICLES_PROBABILITY_RATIOS.length; i++) {
  for (int j = 0; j < Prefs.VEHICLES_PROBABILITY_RATIOS[i]; j++) {
   probs[probs_size-1]=i;
   probs_size--;
  }
 }
}
```

After that, the user is asked to enter the number of vehicles he/she wants to simulate through the console; then a thread pool is created and new vehicle objects, whose type is picked randomly from the previously generated array, are created by a `VehicleFactory` with a delay composed by a fixed value plus a random value picked randomly in a specified rage (both parameters are specified in `Prefs` object). At the end of the simulation, the `ExecutorService` used for thread pool is shutted down, and the simulation terminates; here we can see the code used:

```
private void startSimulation() {
 Scanner scanner = new Scanner(System.in);
 System.out.println("##### PARKING LOT USE CASE SIMULATOR #####\nInsert the number of
vehicles you want to simulate: ");
 int numberOfVehicles = scanner.nextInt();

 ExecutorService simExec = Executors.newFixedThreadPool(numberOfVehicles);
 for (int i = 0; i < numberOfVehicles; i++) {
  int randomType = probs[sr.nextInt(probs.length)];

simExec.submit(vehicleFactory.createVehicle(VehicleType.values()[randomType],type_counters[r
andomType]));
  LOG.info("New vehicle arrived:
"+VehicleType.values()[randomType]+"_"+type_counters[randomType]);
  type_counters[randomType]++;

  try {
   Thread.sleep(Prefs.VEHICLES_BASE_DELAY_MILLS
+sr.nextInt(Prefs.VEHICHLES_DELAY_RANGE_MILLS));
  } catch (InterruptedException e) {
   e.printStackTrace();
  }
 }
 LOG.info("All vehicles have been released.");
 simExec.shutdown();
 try {
  simExec.awaitTermination(Long.MAX_VALUE, TimeUnit.MILLISECONDS);
 } catch (InterruptedException e) {
  e.printStackTrace();
 }
}
```

Operations and mechanisms

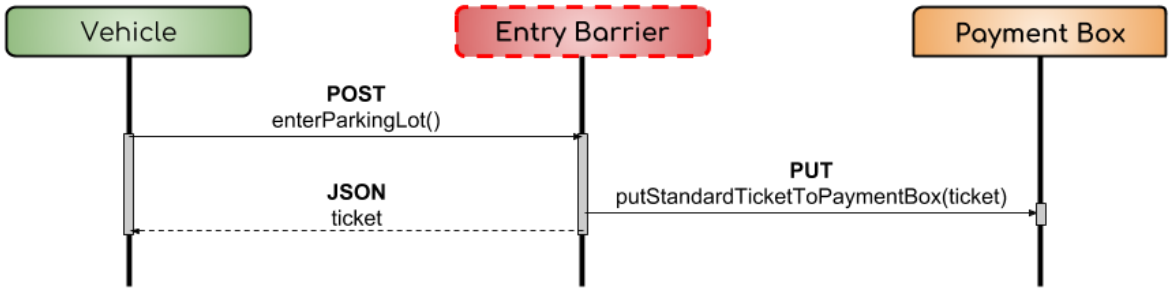
In this section we will see in detail the interactions between all the actors during the vehicle event flow. But before starting, here is a summary diagram of the relations between control devices:



General use case

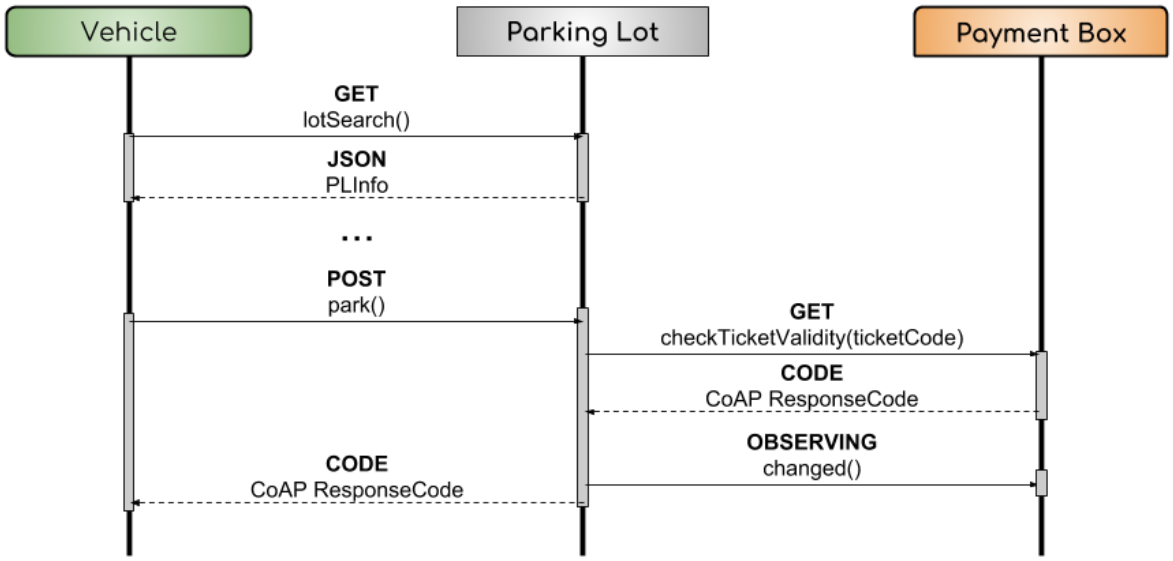
What is described in the following paragraphs is valid for Normal, Moto and Expectant vehicles, while other types have some minor differences that will be treated in the *Variants* section. We will make extensive use of Sequence Diagrams as they are very useful to understand the undergoing mechanisms.

Entering the parking lot



This phase is fairly straightforward: the vehicle makes a request to the entry barrier, which sends the newly created ticket to Payment Box and replies to the vehicle with the ticket formatted in Json.

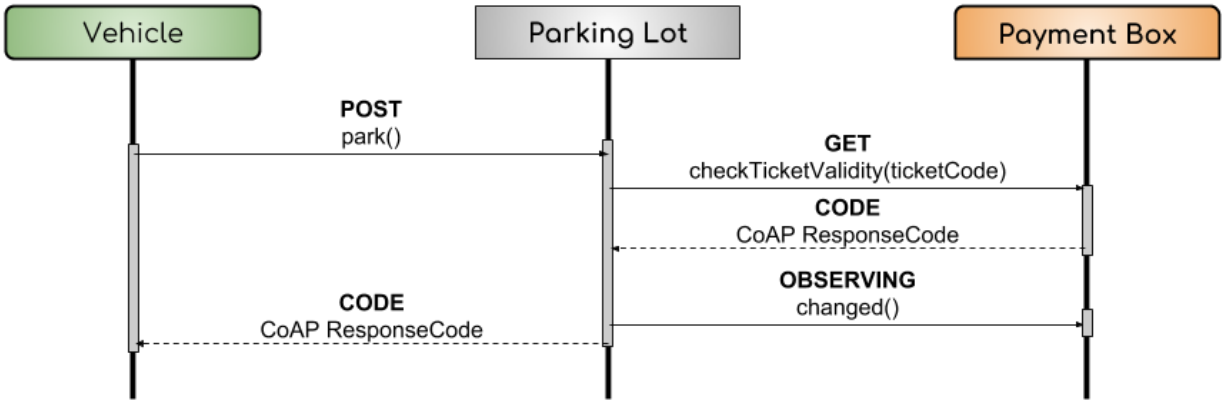
Parking



This section is a bit more complicated. At first the vehicle will repeatedly send GET requests to random parking lots until it finds a good one, then a POST request is performed to that particular parking lot in order to obtain the authorization to park there. The parking lot makes a GET request to the Payment Box to ascertain that the ticket is valid, and in case of a positive match it updates his state (triggering the already established observing relation

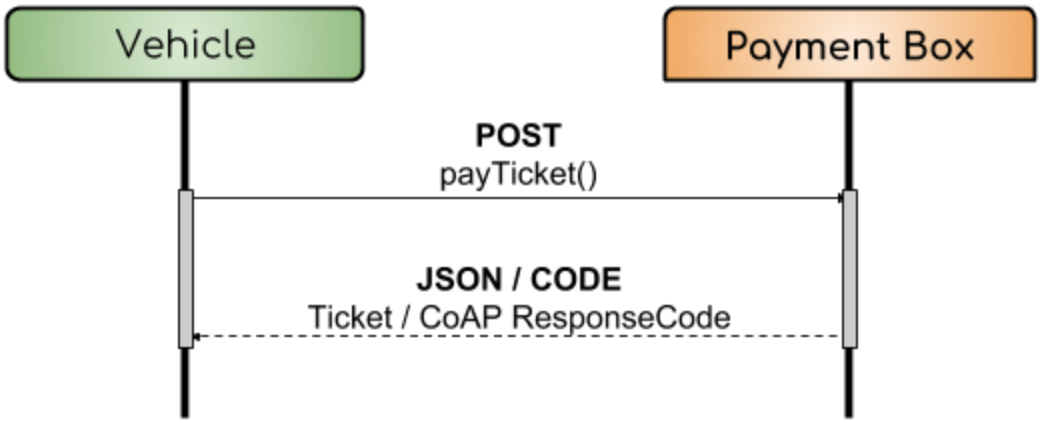
and causing an update of the state also in the payment box) and responding to the vehicle with the appropriate code.

Leaving



The leaving process is almost the same of the parking one: the main difference is not in the sequence of messages but in how the requests are processed by parking lot and payment box, which mechanisms are described in the previous chapters.

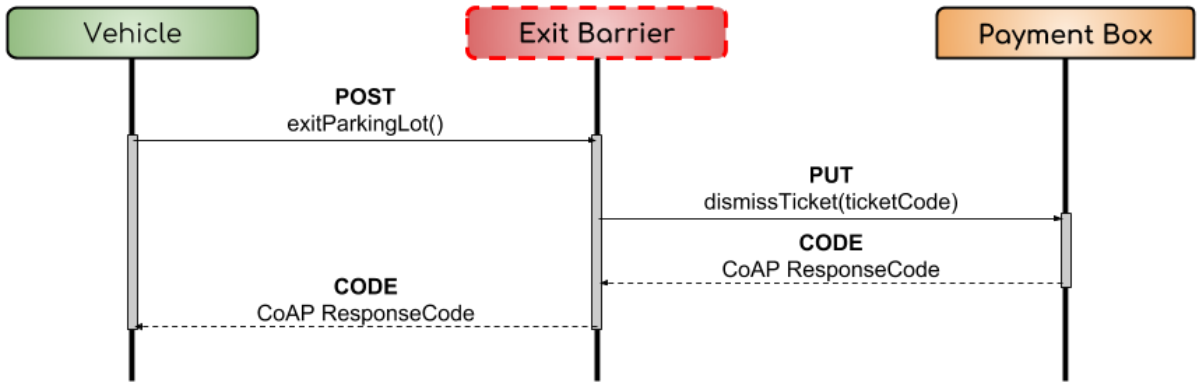
Paying



After the vehicle leaves the parking lot, the next step is going to the Payment Box and pull out the money from the wallet (or use other payment methods): again the sequence of messages is pretty simple, as the core of the process is in the elaboration performed by the Payment Box. If the payment has been successful a copy of the ticket with the new details

is emitted, otherwise a proper response code is emitted depending on what goes wrong during the process.

Exiting the parking lot



At the end, the vehicle makes a POST request to the Exit Barrier, which contacts the Payment Box trying to dismiss the ticket: if the operation is successful, it opens the barrier allowing the vehicle to leave the parking lot, otherwise a proper response code is sent back and the exit is denied.

Variants

Hand

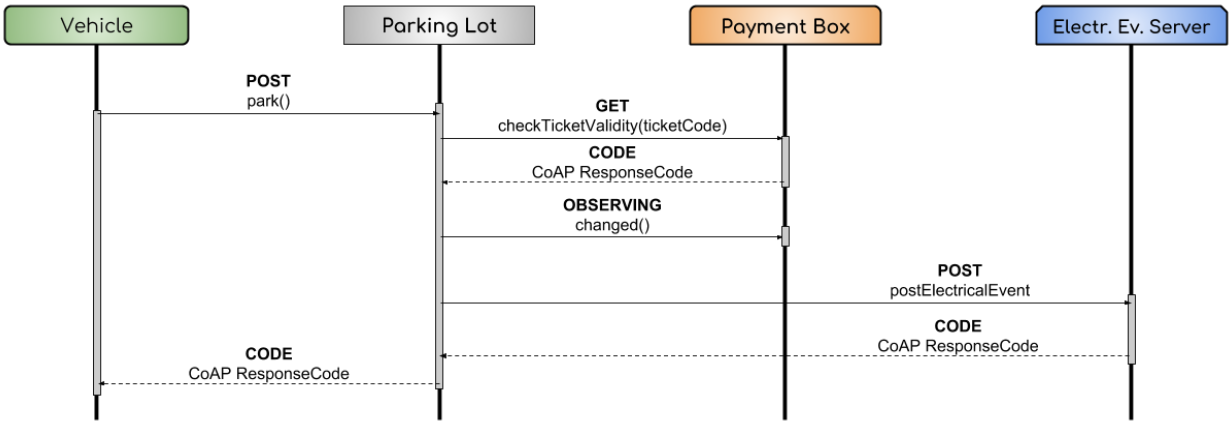
The Hand vehicle sequence in almost the same as the one described above: the only exception is that the Entry Barrier submits the ticket to the Payment Box directly in the Paid list, allowing the vehicle to leave without the need to carry out the payment operation.

Private

Also in this case there is no need to pay anything, exactly like the Hand case; moreover, there is a check of the secret code by the parking lot when the park() and unpark() methods are called, but being an internal operation does not involve changes in the message sequence.

Electrical

Things here are a little bit different, as the parking and leaving operations require an extra step: in this Sequence Diagram is shown the parking process, but the same actions are performed when leaving the parking lot:



The parking lot, in addition to the common operations with the Payment Box, performs a POST to the Electrical Event Server, which task is to maintain a backup of the Electrical Events occurred.

Tests and conclusions

The Smart Parking has been tested several times with the Simulator, using multiple combinations of size, loads, arrival frequencies and more: the system has maintained consistency and robustness in all the situations occurred, even in case of incorrect configuration of the events sequence in vehicles.