GitBerto:
The automated robot guide

Luca Landolfi Luca Marchegiani

July 16, 2022

Abstract

In this paper we propose a small project for the *Mobile Systems* course. The project basically consists of a small robot that is piloted by a mobile device: the user sets a destination and then follows the robot that takes him to the desired location.

We underline that this work is only a very simplified description of the main operation of the system that can be extended in future developments. In this work we only develop e very simple system that shows the basic idea of this project without covering all the aspects of a real *sealable* product, but only those related with the contents of the *Mobile Systems* course with some elements derived by the course of *Ingegneria dei Sistemi Software*.

Contents

1 Introduction and fast Requirement Analysis

Since this is a small project with the aim to experiment something in *Mobile Systems*, we never provide a detailed project with exhaustive analysis. However, we will give some important details by making a fast analysis.

1.1 Main user story

This *user story* shows the main description of the operation of the system:

User Story 1.1: Main User Story

The user:

1. uses his device and **opens the `GitBerto` application** on his mobile that **automatically connects with the paired physical robot**;
2. inserts a destination using the classical *address searching* and the application shows the founded possibilities; then, **selects the desired target** to arrive to and the application calculates the route to get the destination;
3. **clicks on the *GO*** button;
4. **follows the physical `GitBerto` robot that started to move** guiding him towards the destination.

Figure 1: Main user story representation

The figure 1 shows a graphical representation of the steps presented in the main user

story.

1.2 Main overview of the architecture of the system

In order to simplify the work, we also make some **assumptions**:

1. angle between routes are only right (90°);
2. routes are all passable only by pedestrians;
3. all others pedestrian shows the robot and avoid collisions with him;
4. the position retrieved by the mobile device is exact without error.

These assumptions let us concentrate with the main focus of this work without caring about others aspects that are mainly related with Artificial Intelligence

As suggested by the assumption 3, as a requirement **all the computation of positions and routes must be performed by the user device**. So, as expected, **the mobile device and the robot must communicate**, exchanging commands and data.

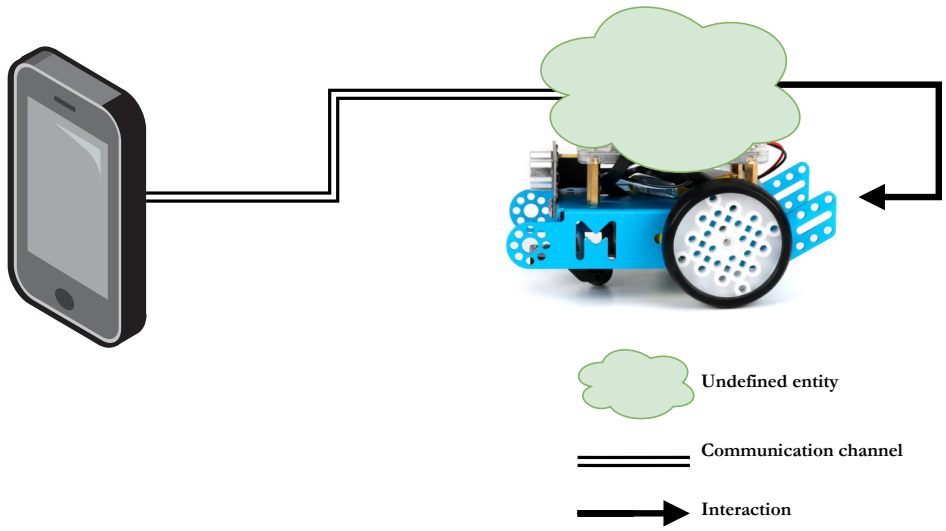

Figure 2: *Coarse* architecture of the system

The first *coarse* architecture is shown by the figure 2 that only say that:

1. there are two ***undefined entities*** that are *executable*: one on the device and the other on the robot;
2. this two entities has a *communication channel* that can be used for **communication**;
3. the entity on the robot can ***interact*** with the physical device.

About the communication, since one of the node is **mobile**, we must choose a mechanism that is supported by this type of node. So, we have restricted possibilities:

- **Bluetooth** that is fully supported from mobile devices (at least from both `iOS` and `Android`);
- **Wi-Fi Direct** that is supported too from the main mobile devices;

- **Internet protocol suite over Wi-Fi**, also supported from all devices.

Since the nodes are mobile, we exclude the set of *Internet protocol suite*. Indeed, this type of communication needs a router in order to be established. Thanks to the modern devices that now all people have, communication over Internet protocol suite implies that user might set up his phone as a Wi-Fi Hotspost, but this could introduce some possible additional costs and resource consumption (especially for battery).

About the mobile devices, is useless to say that the main options are two: `iOS` and `Android`. Since in the course of *Mobile Systems* we have studied `Android`, in this project we build the system considering only it but leaving the possibility to use the competitor in others development.

Finally, about the robot we will consider a physical robot controlled by `Arduino` or Raspberry. We give two main options, but the only requirement is that **the controller of the robot has to support the chosen communication mechanism**. The options we suggest are:

- **mBot**, a robot controlled by an Arduino board;
- **Nano Robot**, intended as a robot kit (like *alphabot*) with a Raspberry Pi mounted as controller.

1.3 Code availability

In the previous *fast requirement analysis* we analysed the devices and the available communication mechanisms. Now, we want to give a fast overview of the code already available. We only consider `Android` for the mobile device.

- about **communication**, Android provides a large set of components and documentation for `Bluetooth` and Wi-Fi Direct; we have also found some Bluetooth libraries that also work for Linux, like `PyBluez` with `Bluez` but no supports for `Wi-Fi Direct`;
- about the **robot** we have some complete supports developed by the professor Antonio Natali for the course of *Ingegneria dei Sistemi Software*, in particular we have the `BasicRobot` system.

We underline that `BasicRobot` is written by using the `QA-System` (also known as `QAK`) that is an infrastructure for **actor meta-modeling and programming**. We have full access to use this infrastructure.

2 Fast problem analysis

First, we have to resolve the main problem that comes from the requirement analysis: we said that the robot has a controller board such as `Arduino` or `Raspberry` and that it must communicate with `Android`.

Notice that even if the robot is a `mBot`, we can easily mount a `Raspberry` over `Arduino`, so we can use all the framework and the technologies that works with `Linux`. Then **we consider that the robot is controlled by a `Raspberry` that is well configured**.

Since we have the BasicRobot already developed and working, we decide to use it in order to build a very simple demo of the system with a mBot with a Raspberry installed over Arduino.

2.1 Connection between Android and Arduino

As shown in the fast requirement analysis, from the Android point of view we have large set of instruments to develop using Bluetooth or Wi-Fi Direct, but we can't say the same thing from the Raspberry (Linux) point of view.

We choose to use Bluetooth for communication because we have found *poor* support for Wi-Fi Direct and also because we think that Bluetooth can consume less.

Even if there are a working Python implementation of Bluetooth for Linux, we decided to make a fast porting of it because the QAK is written in Kotlin.

The entire ported library, called KBluez is fully accessible at this GitHub repository. Basically, there is a Python script (pybluezwrapper.py) launched by the Kotlin library as a system process using Java Runtime: stdin, stdout and stderr of the Python script are captured and handled by Kotlin that is able to send commands to PyBluez and read response by using a proper protocol.

Indeed, pybluezwrapper.py uses the main thread for reading commands from stdin that are sent by Kotlin as json strings. When the Python script receives the command to open a *Bluetooth Socket*, so pybluezwrapper.py launches another thread and entrust the new socket to it associating an unique identifier that Kotlin uses to refer to. **In this way, concurrency is safely guaranteed**, and it's possible to open multiple sockets without blocking the Python script.

The figure 3 shows a general overview of the architecture of our KBluez.

The figure 4 shows the class diagram of the BluetoothSocket interface that is the main key point of our library. Other classes will not be shown at the moment. This interface is implemented by PyBluezSocket class but the implementation will not be exposed in deep.

At the operational point of view, the developer can obtain a BluetoothSocket with RFCOMMby invoking:

```
1 val btSock = KBLUEZ.requestNewSocket(BluetoothServiceProtocol.RFCOMM)
```

Then, this socket can be used exactly as the internet protocol sockets in order to build a client or a server.

We have used BluetoothSocket implementation in order to create a bluetooth server on Raspberry that make it able to communicate with the device of the user. No library or third-party components are needed in Android.

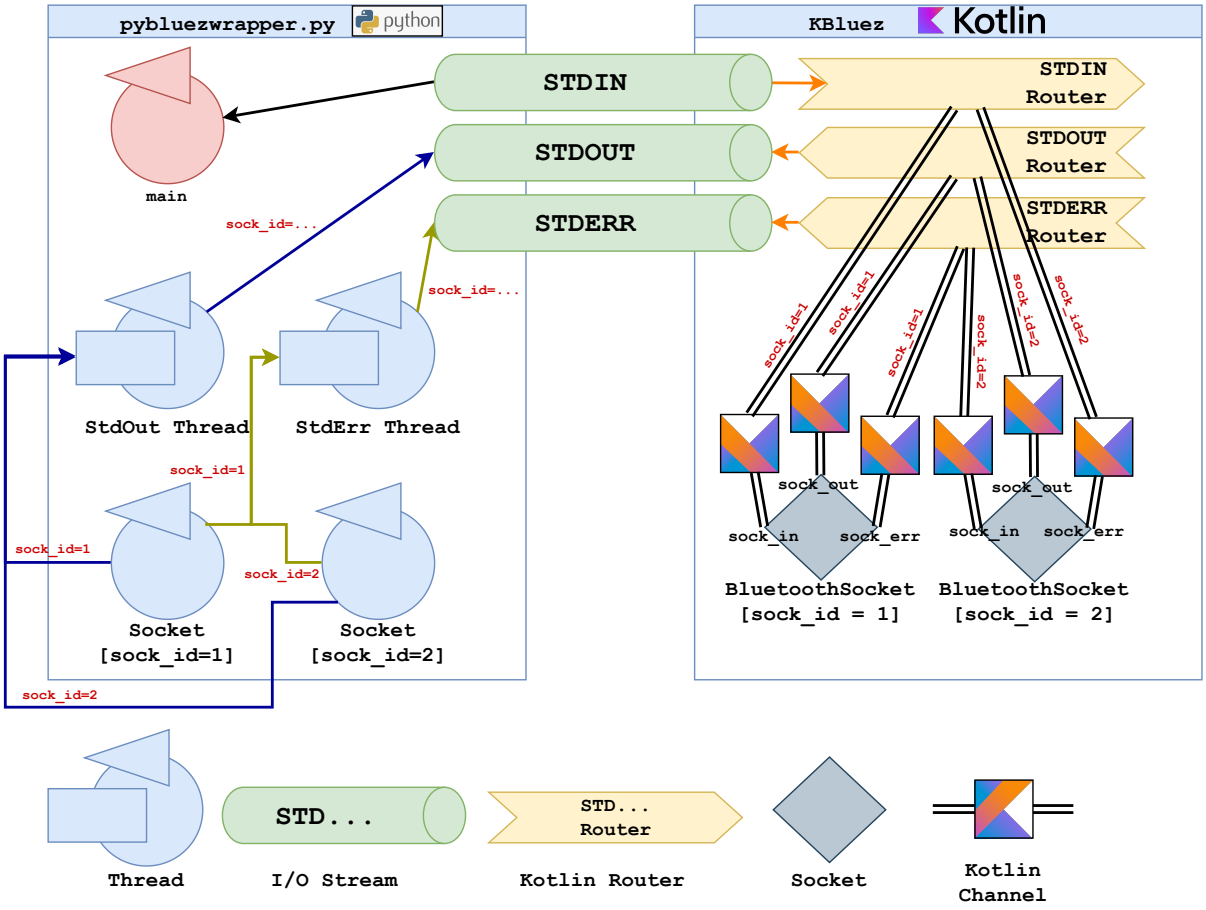

Figure 3: General architecture of the `KBluez` library

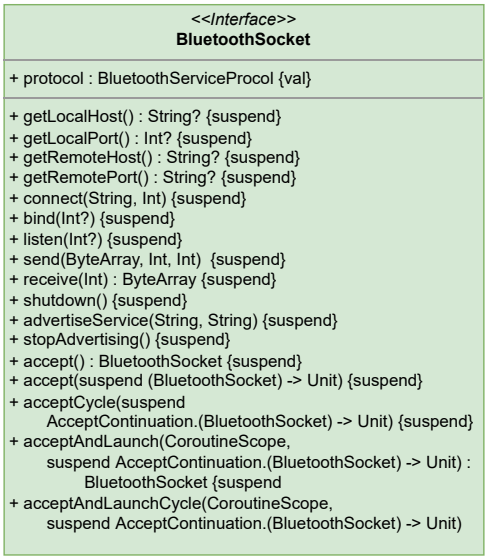

Figure 4: Class diagram of `it.unibo.kBluez.socket.BluetoothSocket`

2.2 Towards Actor Modeling

2.2.1 Running `QAK` infrastructure under `Android`

In the *Mobile System* course we study the fundamentals of `Android` developing. We deeply analysed all components offered to the `Android` developer such as `Activity`, Services,

`Intent`, `Broadcast Received`, `AsyncTask` and so on and so forth...

Now, we want to merge all of these concepts with the already mentioned `actor meta-modeling` and programming. For these purposes, as anticipated, we have the `QA-System` that is well documented at the `Github repository` of the corse of *Ingegneria dei Sistemi Software*.

Just to give a fast overview, **an actor is an active entity that receives messages thanks to a channel that it owns**. So, when an actor receives a message, it performs some action depending on the contents. The `QAK` system make easy to define and handle actors that are mapped into `Kotlin` coroutines and, in addition to this, the infrastructure fully supports the concept of **actors as finite state machine** that change their internal state by receiving messages or *events*.

Using actors and `QAK` let us use *actor meta-modeling* as shown in this example. Once the actor meta-model is well-developed, we can execute it using `QAK`, but this introduces some problems related with the nature of `QAK` that works on normal `JVM`.

1. To define an actor, `QAK` requires it definition in a special `.pl` file that maintains all the defined actors

   > **We have an extension of the entire `QAK` infrastructure made by *Luca Marchegiani* at `QA-Extensions` repository that let the developer use `Annotations` instead of *legacy* configuration file.**

2. To define an actor, in addition to annotations or configuration file, the developer must extend QActorBasic or QActorBasicFsm classes so, since we are using `Kotlin`, multiple inheritance is not allowed than it is not possible to define classes that extend both `Activity` and `ActorBasicFsm`

   > **We have created a new way to define actors that uses the newest interfaces `IQActorBasic` and `IQActorBasicFsm`.**

3. `QAK` creates a log file for each defined actor: this is not easy in `Android` because of the restrictions for writing files

   > **We have added another modification to the extensions that allows to disable file creation in `sysUtil` by adding a simple parameter to the launcher of the `QA-System` (see QAKUtils.kt).**

In addition to this, notice that `Android` **classes are not compiled in `.class` but in `.dex`**, so the annotation scan that is present in the `QA-Extensions` does not work. To quickly resolve this problem, we choose to declare actor classes in a *hard-coded* way inside the QAKUtils.kt. Indeed, for extension purposes, the launcher developed in `QA-Extensions` let the developer to add some parameters, so it is easy to make additional extensions like this.

In future development it is possible to use some more accurate mechanisms such as *build time scanning* of the annotations.

2.2.2 Coroutines in Android with ViewModel

As studied in *Mobile System* course, if an Android developer has to perform a long IO operation inside ad Activity, he should use other components like AsyncTask that, in simple terms, creates a new thread for the *long-time* operation, and *after* calls a function in which the developer can put the UI updates.

AsyncTask are now deprecated in favour of the new Kotlin concurrency utilities (see *Kotlin* coroutines components in Android).

Before we knew this, we had already decided to experiment something new such as ViewModel that is a component *designed to store and manage UI-related data in a lifecycle conscious way* and that can be used in combination with Kotlin coroutines.

Even without ViewModel, in Kotlin it is easily possible to execute pieces of concurrent code that update UI inside a coroutine. For example, suppose that you have to launch a long IO operation that loads your recipes stored into a database and have to show it on the activity:

```
class BluetoothConnectionActivity : AppCompactActivity() {

 lateinit var tableLayout : TableLayout
 val scope : CoroutineScope = ...
 val recipesRepo : RecipesRepo = ...

 override fun onCreate(savedInstanceState: Bundle?) {
  tableLayout = findViewById(R.id.recipesTableLayout)
  scope.launch {

   //retrieveAll() method uses blocking network IO
   //operations such as read from remote database
   val recipes = recipesRepo.retrieveAll()

   withContext(Dispatchers.Main) {
    for(recipe in recipes) {
      //update tableLayout with recipe...
    }
   }
  }
 }
}
```

Indeed, the launch primitive *launches* a new coroutine that is dispatched to another thread (or to a pool of threads), but when we called Dispatchers.Main then the block inside is passed to the *main* thread that executes it. So, *UI* updates are performed.

As suggested by *Android* developer guide, instead of this pattern it is better to use ViewModel that automatically manage the lifecycle of the coroutines launched inside this. We have extended the suggested solution with this pattern:

```
 class RecipesIO(private val recipesRepo : RecipesRepo) {

  suspend fun retrieveAll() : Result<List<Recipe>> {
   return withContext(Dispatchers.IO) {
    return@withContext try {
```

```
      Result.success(recipesRepo.retrieveAll())
    } catch (e : Exception) {
      Result.failure(e)
    }
   }
  }

 }

class RecipesViewModel(private val recipesIO : RecipesIO) {

 private val onRetrieved = mutableListOf<
   suspend (Result<List<Recipe>>) -> Unit>()

 fun addOnRetrieved(uiUpdate : suspend (Result<List<Recipe>>) -> Unit) {
  onRetrieved.add(uiUpdate)
 }

 fun asyncRetrieveAll() {
  viewModelScope.launch{
   val recipes = this@RecipesViewModel.recipesIO.retrieveAll()
   onRetrieved.forEach { uiUpdate ->
    uiUpdate(recipes)
   }
  }
 }

}

class BluetoothConnectionActivity : AppCompactActivity() {

  lateinit var tableLayout : TableLayout
  val recipesRepo : RecipesRepo = ...
  val recipesViewModel = RecipesViewModel(recipesRepo)

  override fun onCreate(savedInstanceState: Bundle?) {
   tableLayout = findViewById(R.id.recipesTableLayout)
   recipesViewModel.addOnRetrieved { result ->
     result.onSuccess { recipes ->
      for(recipe in recipes) {
       //update tableLayout with recipe...
      }
     }
     result.onFailure { exception ->
      //Notify failure
     }
   }
   recipesViewModel.asyncRetrieveAll()
  }
 }
```

Notice that `viewModelScope` is a special `CoroutineScope` owned by each `ViewModel` that automatically cancel all coroutines launched inside it if the related owner is cancelled. So, using it is totally safe!

`Android` let the developer use another special scope that is `LifecycleScope` (see the documentation for details).

This mechanism is very powerful but **the developer must pay attention and be careful** because if he exaggerates he should saturate the main thread if he forgets to use `Dispatcher.IO` for blocking operations.

2.2.3 From `Coroutines` to `QActors`

We have just shown that an `Activity` can launch asynchronous operations using coroutines that can also update the `UI`. So, what could happen if we *transform* an `Activity` into a *finite state machine actor*?

Indeed, we added some modifications to the `QA-Extensions` that let the developer define actors **using an interface** thanks to the `Kotlin` **delegates** that let to make something similar to the multiple inheritance.

We create the class `ActorAppCompactActivity` that automatically *configure* the actor nature of an activity. A class that extends `ActorAppCompactActivity` is an `Android` activity that also is a finite state machine actor. The developer must however remember to use the proper annotations on his class[a].

[a]Notice that for instantiation reasons the actors that extends `ActorAppCompactActivity` must be annotated with `StartMode(TransientStartMode.MANUAL)`

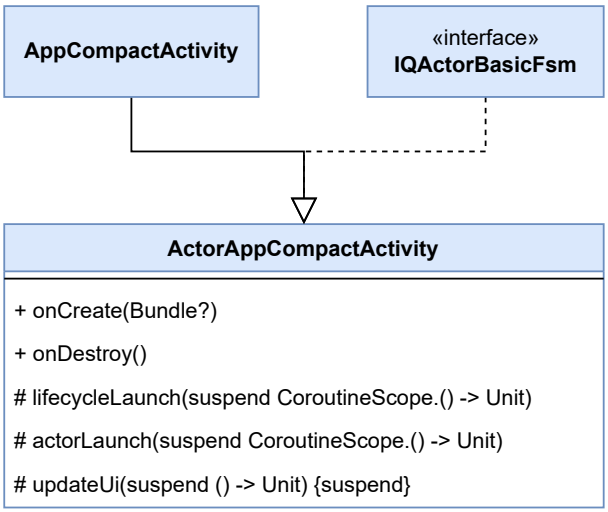

Figure 5: Class diagram of `ActorAppCompactActivity`

All the classes that extends `ActorAppCompactActivity` **must** also use th `by` keyword offered by `Kotlin` in order to get the implementation of the `IQActorBasicClass` following this pattern:

```
1 val APP_SCOPE = CoroutineScope(SupervisorJob() + Dispatchers.Default)
2 val DEFAULT_PARAMS = mutableParameterMap()
3 .addBlockIOParam()
4 .addAnnotatedClassesParams(
5     MyActivity::class.java
6 )
7 .addSystemScope(APP_SCOPE)
8
```

```
@QActor("my_ctx")
@StartMode(TransientStartMode.MANUAL)
class MyActivity : ActorAppCompactActivity(), IQActorBasicFsm
        by qakActorFsm(MyActivity::class.java,
        Dispatchers.Default,
        DEFAULT_PARAMS)
{
   ...
}
```

If you want to add other activities, you have to add the relative class into `addAnnotatedClassesParams()` method that builds the `ParameterMap`. In addition to this, the class must be passed also in the `qakActorFsm()` method after the `by` keyword: unlucky, this code repetition is needed because we are in `Android` so we can use annotation scanning as said.

Notice also the `APP_SCOPE`: this scope is created in order to avoid the use of `GlobalScope` that is strongly discouraged.

To conclude this overview, we underline these methods:

- `lifecycleLaunch(block : suspend CoroutineScope.() -> Unit)`
  launches a coroutine that executes the passed `block` using the `LifecycleCoroutineScope` associated with the lifecycle of the activity;
- `actorLaunch(block : suspend CoroutineScope.() -> Unit)`
  launches a coroutine that executes the passed `block` using the `CoroutineScope` associated with the actor;
- `updateUi(block : suspend () -> Unit)`
  a `suspend` function that executes the passed `block` using `Dispatchers.Main`; this method can be used from a coroutine in order to perform `ui` updates.

In order to prevent the overload of main thread, we decide to use the `Dispatchers.Default` as default dispatcher of all coroutines in which actor are executing, so, if an *activity actor* have to perform `ui` updates, he can call `updateUi` method. This is the reason why we are using the `APP_SCOPE` that is created using `Dispatchers.Default`.

2.2.4 Logical architecture

As final result of this very fast analysis, we provide the **logical architecture** of the `GitBerto` system in the figure 6.

We can also provide a small description of all the actor that will be present in the `Android` application of `GitBerto` system:

- **BluetoothActivity**
  the activity that makes the user device able to establish a connection with the robot;
- **MainActivity**
  the main activity that user can use to set the destination and start the trip;

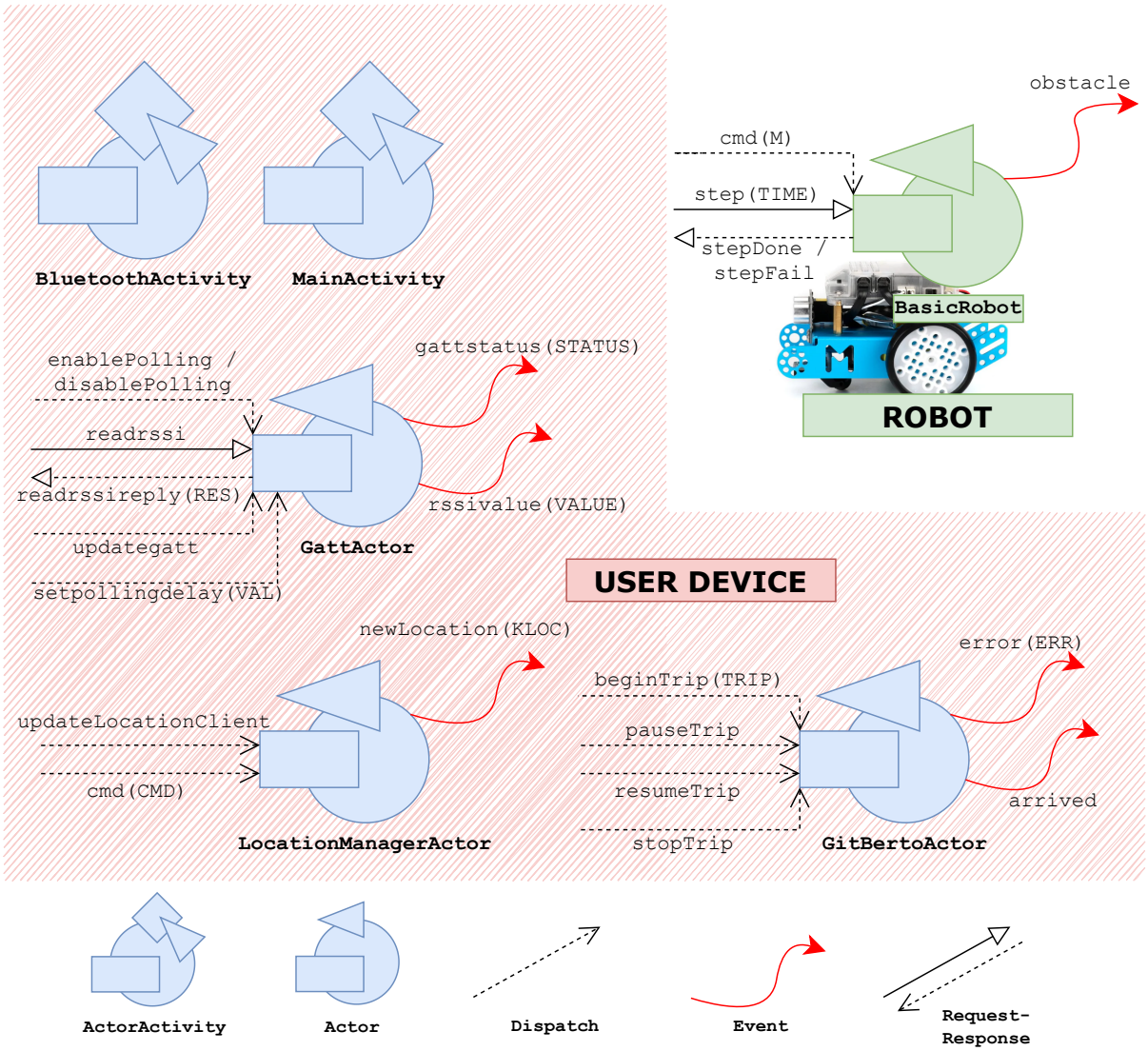

Figure 6: Logical architecture of `GitBerto` system

- <u>GattActor</u>
  the actor that uses `Bluetooth Low Energy` for `GATT` functionality in order to continuously check the presence of the robot when the connection is established;
  - `enablePolling` and `disablePolling`: dispatches that enable and disable the polling work of the actor;
  - `setpollingdelay`: dispatch that set the polling delay of the actor;
  - `readrssi`: request that asks the actor for reading the current `RSSI` value; the actor replies with the `readrssireply` reply with the readed value;
  - `updategatt`: a dispatch that invites the actor to update the `BluetoothGatt` he is using;
  - `gattstatus`: an event that is fired on a change of the status of the related `BluetoothGatt`;
  - `rssivalue`: an event that is fired when an `RSSI` value is readed;

- **LocationManagerActor**
  the actor that *manages* the location sending events when user move changing his position;
  - `updateLocationClient`: a dispatch that invites the robot to update the *location provider* he is using to perform location updates; this message can be sent only once when this actor is started in order to *set up* it;
  - `cmd(CMD)`: a dispatch that can be used to enable the *location monitoring* work of this actor; the possible arguments of this message are two: `enableMonitoring` or `disableMonitoring` to enable or disable;
  - `newLocation`: an event that is fired when the location of the device is changed; the argument of this message is a `JSON` string that serialize a `KLocation`[1] instance;
- **GitBertoActor**
  the actor that make the system able to interact with the robot using the *high level* operations that let to manage the trip execution;
  - `beginTrip(TRIP)`: a dispatch that start the work of the robot as a *trip guide*; the argument of this message is a `JSON` string that serializes a `Road` object that maintains the road of the entire trip;
  - `pauseTrip` and `resumeTrip`: dispatches that respectively pause the current trip of the robot and resume it;
  - `stopTrip`: a dispatch that can be used in order to *cancel* the trip stopping the robot;
  - `error(ERR)`: an event that is fired when there is an error; the argument of this message is a string description of the thrown error;
  - `arrived`: an event that is fired when the robot is correctly arrived at the destination.

3 Fast project

The full code of the `Android` application is available on this `GitHub` repository: *link*.
The full code of the `kBluez` library is available on this `GitHub` repository: link.

3.1 Project of the `Bluetooth` communications

As said in the fast analysis, `Bluetooth` is the chosen communication technology to realize the interaction between robot and user's device. Android developer documentation fully explain how to set up `Bluetooth` while `kBluez` is made by us to make this project work.

About `Android`, as suggested by the documentation, the main operation to *connect* to a device is:

- **find** the device by executing a *discovery* operation;

[1] We have suitably created this class because the `Location` that is present in `Android` is not serializable.

- **connect** to the discovered device obtaining a `BluetoothSocket`;
- **exchange** data throw the socket.

Notice that differently from the normal *INET* socket, the `Bluetooth` connection can be established only using the `UUID` of a `RFCOMM` service.

In addition to this, in `Android` some operations are natively realized with some asynchronous mechanism such as `IntentFilter` or `BroadcastReceiver`. The obvious motivation for that is to prevent some *unconscious* use of blocking `IO` operations on the main thread that with this mechanism is avoided.

Even if this way *to do the things* should more safe, it introduces some complications especially for the legibility of code: indeed the developer must register *receiver* that will be activated when required the action has been performed, like imposed by `Intent` or `BroadcastReceiver`.

Then, consider if you want to perform a discovery operation using bluetooth in order to search for devices, storing the results in a collection. Following the android guide, the developer must use this pattern:

```
//The collection to store results
val devices = ConcurrentLinkedDeque<BluetoothDevice>()

override fun onCreate(savedInstanceState: Bundle?) {
 ...

 // Register for broadcasts when a device is discovered.
 val filter = IntentFilter(BluetoothDevice.ACTION_FOUND)
 registerReceiver(receiver, filter)

 //Start the discovery
 bluetoothAdapter.startDiscovery()
}

// Create a BroadcastReceiver for ACTION_FOUND.
private val receiver = object : BroadcastReceiver() {

 override fun onReceive(context: Context, intent: Intent) {
  val action: String = intent.action
  when(action) {
   BluetoothDevice.ACTION_FOUND -> {
    // Discovery has found a device. Get the BluetoothDevice
    // object and its info from the Intent.
    val device: BluetoothDevice =
    intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE)
    devices.add(device)
   }
  }
 }
}

override fun onDestroy() {
 super.onDestroy()
 ...

 // Don't forget to unregister the ACTION_FOUND receiver.
 unregisterReceiver(receiver)
```

```
}
```

We could make this code more *elegant* using the smart notation of `Kotlin` lambdas, but however asynchronism could confuse the developer. In addition to this, if the operation are nested (for example, *if a device is found that connect to it*) then code inevitably ends with a *storm* of lambdas which can be very illegible.

For this reason, we decide to provide a solution that restore the synchronism by using the simple class `BluetoothDiscoveryBroadcastReceiver` and coroutines:

```
private val resultChan = Channel<Any>()
private val bluetoothDiscoveryBroadcastReceiver =
      BluetoothDiscoveryBroadcastReceiver()
private val controllerOnDeviceDiscovered : (BluetoothDevice) -> Unit =
{ device ->
 scope.launch {resultChan.send(device)}
}
private val controllerOnDiscoveryFinished : () -> Unit =
{
 scope.launch {resultChan.send(BluetoothAdapter.ACTION_DISCOVERY_FINISHED)}
}

suspend fun discoverDevice(activity : Activity,
       onDeviceDiscovered : (BluetoothDevice) -> Unit = {}) :
         Set<BluetoothDevice> {
 bluetoothDiscoveryBroadcastReceiver
    .addOnDeviceDiscovered(onDeviceDiscovered)
 bluetoothDiscoveryBroadcastReceiver
    .addOnDeviceDiscovered(controllerOnDeviceDiscovered)
 bluetoothDiscoveryBroadcastReceiver
    .addOnDiscoveryFinished(controllerOnDiscoveryFinished)

 withContext(Dispatchers.Main) {
  val actionFoundFilter = IntentFilter(BluetoothDevice.ACTION_FOUND)
  activity.registerReceiver(bluetoothDiscoveryBroadcastReceiver,
          actionFoundFilter)
  val discoveryFinishedFilter = IntentFilter(
        BluetoothAdapter.ACTION_DISCOVERY_FINISHED)
  activity.registerReceiver(bluetoothDiscoveryBroadcastReceiver,
          discoveryFinishedFilter)
  bluetoothAdapter!!.startDiscovery()
 }

 val devices = mutableSetOf<BluetoothDevice>()
 var res : Any
 do {
  res = resultChan.receive()
  when(res) {
   is BluetoothDevice -> {
    devices.add(res)
   }
  }
 } while (!(res is String &&
     res == BluetoothAdapter.ACTION_DISCOVERY_FINISHED))

 withContext(Dispatchers.Main) {
  activity.unregisterReceiver(bluetoothDiscoveryBroadcastReceiver)
 }
```

```
49  bluetoothDiscoveryBroadcastReceiver
50      .removeOnDeviceDiscovered(onDeviceDiscovered)
51  bluetoothDiscoveryBroadcastReceiver
52      .removeOnDeviceDiscovered(controllerOnDeviceDiscovered)
53  bluetoothDiscoveryBroadcastReceiver
54      .removeOnDiscoveryFinished(controllerOnDiscoveryFinished)
55
56  return devices
57 }
```

Notice thatndiscoverDevice is a method that is executed by a coroutine that waits for the discovery operation completion. Theoretically, this method should introduce only a very small overhead caused by the `scope.launch` that is present by the lambda that are invoked from the `BroadcastReceiver` of the bluetooth. Indeed, the suspension of calling coroutine on the result channel should introduce no additional costs.

In future implementations, this function might be refined in order to be called also from coroutines that are executing on the main thread by using the appropriate dispatcher.

Other Bluetooth operations also use this pattern so the figure 7 shows the `UML` diagram of a *controller* for the Bluetooth that can be used from `BluetoothActivity`.

The figure 8 shows the state diagram of the `BluetoothActivity`. Notice that **the method `setupBluetooth` of the controller must be called into the `onCreate` method of `Activity` in order to make it work**.

About the states in `BluetoothActivity`:

- **begin**:
  in this state the actor prepare itself for doing the Bluetooth operations and showing results.
- **setupBluetooth**:
  the actor check if Bluetooth is already been set up by the activity: if not, the actor goes into **unableToSetupStatus** that only shows to the user a toast saying that it is not possible to use Bluetooth.
- **connectToSavedDevice**:
  the actor tries to connect the *saved device* that is the last correctly used by the application.
- **scanForNewOnes**:
  the actor uses the controller to scan for devices showing the found in real time. When the user selects a device, the actor immediately transit to **connectToSavedDevice** by epsilon move.

We also underline that the actor waits for a `setupBluetooth` message from its activity in order to perform Bluetooth operations: this allows to set up Bluetooth from the `onCreate`, notifying the actor when this operation is concluded, so the actor can correctly use a well initiated controller.

Figure 7: UML diagram of `BluetoothController`

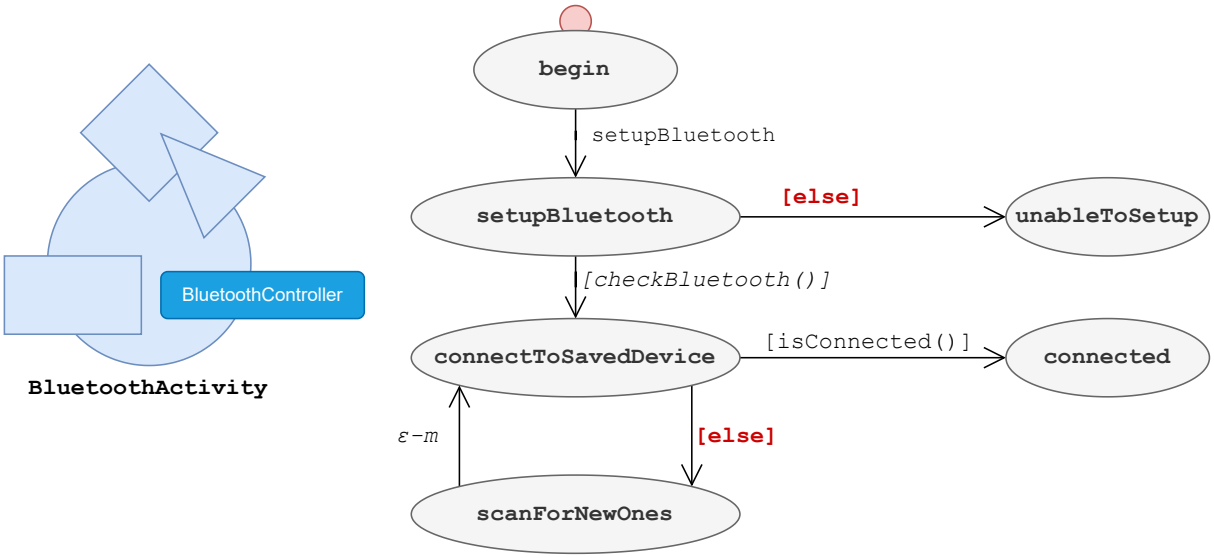

Figure 8: State diagram of `BluetoothActivity`