



# SPRINT 4

23.10.2020

---

Laura Mazzuca, Nicholas Antonio Carroll, Giuseppe Giorgio

## Requirement Analysis

### Assumptions

For this Sprint we assume that:

- we **will** consider the presence of **multiple clients**;
- The client, the barman and the smartbell will be simulated and grouped together with the waiter context-wise;
- we **will** consider the existence of the manager and his interface;
- we **will not** take into account optimization of tasks;
- The waiter **will** be able to interrupt the cleaning task;
- We **will** take into account the MaxStayTime and MaxWaitTime timeouts.

*Additional requirements form our Dev Team:*

- Address the waiter's collisions with obstacles to correct the course.

### Manager Interface

In this Sprint we are covering the introduction of the Manager interface. During the first SPRINT analysis, we identified the following information:

- Waiter position;
- Waiter current task;
- Table state, as in available, dirty or busy.

As we analyzed deeper the requirement, we have identified some more information the manager should be able to check and that evidently represent more accurately the current state the tearoom is in:

- How many tables are currently free;
- If there's a client waiting to enter or not;
- How long the client is staying at the table;
- Information about the smartbell's temperature measurements;
- What is the barman doing and what is he preparing at the moment.

## Problem Analysis

### System Architecture

The Architecture, at this point, is unchanged from the previous sprint.

### Waiter Improvements

#### Handling Multiple Clients

In this SPRINT we finally address the problem of handling more than one client in a more structured way.

First of all, we need to separate the contexts of Smartbell and Waiter in `ctxsmartbell` and `ctxwaiter`.

The Waiter was developed almost completely ready to handle more than one Client.

What it's missing is the management of the Client admitted by the Smartbell, but waiting to enter the Tearoom. The solution to this problem requires the Waiter to remember the fact that there is a waiting Client and the possibility for it to go get him at the entrance door when a table is freed and sanitized.

#### MaxWaitTime

The MaxWaitTime value is sent to the Client who is accepted by the Smartbell, but there are no tables sanitized ready for her to sit. During this time, if a table frees up and gets cleaned in time by the Waiter, the Client can be deployed to it, otherwise, when the timeout expires, the Client has to leave the Waiting Hall.

To handle this in the best way the Waiter should be improved to be able to know:

- If there are any waiting clients
- How many are there
- In which order they arrived

#### MaxStayTime

When the MaxStayTime expires the Waiter should "force" the Client to pay and leave. This functionality should be implemented by the Client when the Waiter brings the drink at the table. The amount of time will be saved in the Waiter and communicated to the Client together with the delivery of the drink.

#### Atomic Actions turned Interruptible

The action of cleaning the table is not as important as receiving other requests so it will be made interruptible. When a request with higher priority arrives, the waiter will stop cleaning and take care of it. When he is done he will return to cleaning. The only exception, obviously, is when a Client is waiting outside and there are no cleaned tables.

## Collisions

To address the collisions problem, we can use to our advantage the fact that when the Virtual Robot collides, it knows what kind of obstacle it was.

This will obviously work only with the Virtual Robot, as of now, but the solution can be extended to work with a robot that has a Smart Webcam that can recognize what kind of obstacle it has in front of it and categorizes them with the same ids.

The idea is that, since we know that the obstacles are static, we know that they have a fixed position. If we enhance the WaiterWalker's knowledge base to know, depending on what it hit and in what direction it is, where it actually is on the map, we can correct the current position and resume the interrupted movement.

In our case, we have that the **actual position** can be given by:

Obstacle ID \ Direction	upDir	downDir	leftDir	rightDir
<b>table1</b> [pos = (2, 3)]	(2, 4)	(2, 2)	(1, 3)	(3, 3)
<b>table2</b> [pos = (4, 3)]	(4, 4)	(4, 2)	(3, 3)	(5, 3)
<b>wallUp</b> [pos = (X, -1)]	(X, 0)	\	\	\
<b>wallDown</b> [pos = (X, 5)]	\	(X, 4)	\	\
<b>wallLeft</b> [pos = (-1, Y)]	\	\	(0, Y)	\
<b>wallRight</b> [pos = (7, Y)]	\	\	\	(5, Y)

Where X and Y are the current X and current Y the waiter is at.

## Manager Interface

Building on top of the architecture we have adopted, the Controller will handle the interaction with the Model, the Tearoom Context, as a CoAP resource, listening for updates of the Resources and redirecting them to the listening Web Interface.

## Tearoom System's CoAP Updates

At this point, we want these updates to be **easily analyzable** by whomever receives them, so we need to introduce a **precise structure for the payload** that should be parameterized depending on the actual task each resource is executing.

The information that a message should be able to contain is:

#### Waiter

- **Busy:** is the waiter waiting for a task to do or is he waiting?
- **ClientID:** current Client being served.
- **Table:** current Table being served.
- **Order:** contains the kind of drink the Client has ordered, if he's handling a requestOrder, or the drink the Waiter is bringing to the Client.
- **Payment:**
  - if the Waiter is telling the Client how much he has to pay, it contains the price asked to the Client;
  - if the Client has paid, it contains the amount paid by him.
- **WaitTime:** it contains the time the Client has to wait upon arrival.
- **MovingTo:** contains the destination of the movement the Waiter is performing.
- **MovingFrom:** contains the starting point of the movement the Waiter is performing.
- **ReceivedRequest:** contains the kind of request for a task to perform the waiter has received by the Client. They can be either:
  - Order
  - Pay
  - DeployEntrance
  - DeployExit
- **Arrival:** contains the information on where the client has arrived when handling a task.
- **AcceptedWaiting:** contains information on letting in a client that is waiting outside.
- **TableDirty:** contains information on the state of the table.

#### Walker

- **Position:** the position the Waiterwalker is currently in in terms of X,Y.

#### Smartbell

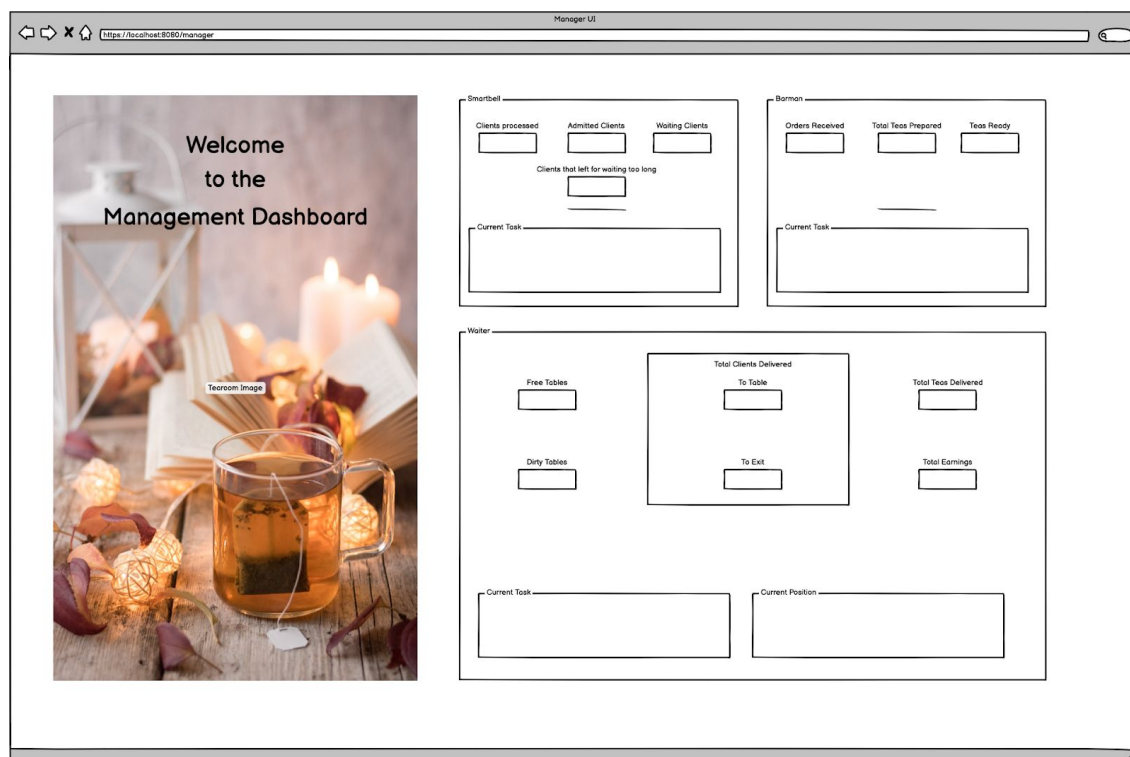
- **Busy:** if the Smartbell is currently handling a Client or not.
- **ClientArrived:** when a Client rings the Smartbell.
- **ClientAccepted:** when a Client has been accepted (temperature  $\leq 37.5^{\circ}$ ).
- **ClientDenied:** when a Client has been denied access and is asked to leave (temperature  $> 37.5^{\circ}$ ).

## Barman

- **Busy:** if the Barman is currently preparing a drink or not.
- **PreparingForTable:** table ID for which the Barman is currently preparing a Drink.
- **PreparingOrder:** Drink the Barman is currently preparing.
- **OrderReadyTable:** table ID for which the order is ready.
- **OrderReady:** drink ready.

## Interface Mockup

The following picture is intended as a guideline of what the Interface should look like to enable the Manager to have, at a glance, all the information required.



## Test

As an assumption for these tests, we take for granted that the Clients CANNOT request tasks in the wrong order.

## Tearoom System with Messages

- Received message has the right number of payloadArgs;
- PayloadArgs in the received message are of the right type;
- Received messages should be coherent with the requested task.

## Multi Client Handling with Messages



For each client we check that:

- Received message has the right number of payloadArgs;
- PayloadArgs in the received message are of the right type;
- Received messages should be coherent with the requested task.

Then, regarding concurrency we check that:

- The second Client that come in the tearoom is brought to the other table wrt the first Client;
- The assigned Client IDs are all different;
- The Client waiting in the hall either runs out time and exits the building or is seated on a sanitized table.

## Project

### Waiter Adjustments

#### Tearoom System's CoAP Updates

The CoAP updates are done in a JSON format, by encapsulating the state of each actor in a Class to make sure that only the correct data is inserted in the state through a number of set methods and that it's easy to add new functionalities. These classes use a kotlin JsonObject to organize the data and have a function to transform it in a JSON format to send with an updateResource in the qak files.

The classes contain the following attributes:

##### WaiterJson.kt

- Busy: Boolean
- ClientID: String
- Table: int
- Order: String
- Payment: Boolean
- WaitTime: int
- MovingTo: String
- MovingFrom: String
- ReceivedRequest: String
- Arrival: String
- acceptedWaiting: Boolean
- tableDirty: Boolean

##### WalkerJson.kt

- PositionX: int
- PositionY: int

##### SmartBellJson.kt

- Busy: Boolean
- ClientArrived: int
- ClientAccepted: int
- ClientDenied: int

##### BarmanJson.kt

- Busy: Boolean
- PreparingForTable: int
- PreparingOrder: String
- OrderReadyTable: int
- OrderReady: Boolean

Since we have multiple actors sending CoAP updates and since a Client might want to subscribe as well as exchange direct messages to one Actor or another, we have split the tearoom context into four different contexts to better process the updates:

- **ctxtearoom** [localhost:8072]: contains the actors Waiter and WaiterWalker
- **Ctxbarman** [localhost:8070]: contains the actor Barman
- **Ctxsmartbell** [localhost:8071]: contain the actor Smartbell
- **Ctxwalker** [localhost:8050]: contain the actor Walker



## Waiting Clients and Update to Enter the Room

To address the problem we face when multiple clients are waiting to get in the room, we need **a structure to keep the order of the clients** that have arrived, so as to send a CoAP update to the first Client that has been waiting to let her know she can stop the countdown and wait for the Waiter to arrive and deploy her to the table.

The best way to do this is to **implement the structure as a queue of Client IDs**, this way, once a table has been cleaned up, we send a CoAP update to let the listeners know which client should enter next. The Server will then handle the rest by forwarding the information to the correct Client thanks to the Client-UUID map we will talk about in the paragraph “Refactoring of the Server”.

## Atomic Actions turned Interruptible

The action of cleaning the table is made interruptible, working on it for 0.5 seconds and then sending a dispatch to himself containing the command to continue cleaning and returning to the listening state.

If there are no other requests in queue the waiter will immediately go back to the table cleaning state and continue working, if there are other, more urgent, requests he will take care of them before returning to cleaning the table. The amount of work already done to clean the table is saved in a variable for each table, so that the waiter does not have to start over.

If both tables are dirty, the Client doesn't go back and forth from one table to the other, but finishes first the table that's almost completely cleaned and then moves to the other one. This is handled by checking which was the last table cleaned, if any, and for how much time of it was done. If it's not yet clean, we resend the message for the table that was requested, so as to not lose the input to go clean it, and then we go finish the last table.

This might be handled better by checking with the KB every so much time, but doing this with messages enables us to react to a dirty table right away.

## Handling the Waiter's Collisions

To implement the idea explained in the Problem Analysis, we first need to make some modifications to the **basicrobot**, since, as of now, the event emitted when hitting an obstacle does not contain the obstacle's ID.

To change that, we need to modify:

1. the classes handling the pipeline from the event emitted by the virtual robot to the basicrobot actor;
2. The event message emitted by the basicrobot;

3. The Walker to handle what happens when a step fails because it hit that specific obstacle;
4. The WaiterWalker to redirect the Walker accordingly to its knowledge of the positions and the map.

### Adjusting the pipeline

What was missing in the pipeline was the forwarding of the collision cause, aka the obstacle name. To change that we had to change the event message emitted by the `sensorObserver(it.unibo.qak20.basicrobot.resources.robotVirtual.virtualrobotSonarSupportActor.kt)` to not only contain the distance but also the object ID of the collision cause. The event is now defined as:

```
Event sonar : sonar( DIST, CAUSE )
```

And is now forwarded with all of its content by the `rx` pipeline to the `baiscorobot` with the event `obstacle`, which now also contains both `DIST` and `CAUSE`.

### Inside the basicrobot

Here, what we had to change was the event emitted when a step fails; it is now defined as:

```
Event obstacle : obstacle( ARG1, ARG2 )
```

And it now returns both the distance and the obstacle ID.

### Changes to the Walker

In our system, it is the Walker that interacts with the `basicrobot`, so it's also the one that receives the aforementioned event. When it does, it now handles the `stepFailed` state by sending a reply to the initial `doPlan` message in the form of:

```
Reply walkerError : walkerError( CURX, CURY, CAUSE, CURDIRECTION )
```

And then goes in the state `WaitReq` to wait for the upper layer to tell it what to do. In this state, if the Walker receives the new `posCorrection` message, defined as:

```
Request posCorrection : posCorrection(X,Y)
```

We enter the `correctAndResume` new state. The `posCorrection` sends to the Walker the X and Y coordinates of the position it's actually in, depending on the information about the collision that was forwarded to the layer handling this information. To adjust the map, the Walker performs the following steps:

1. Computes the plan to get from the current position to the position received from the upper layer;
2. Executes the plan *virtually*, that is it only updates the map with the moves but does not actually request them to be performed by the `basicrobot`. An exception is made for the `l` and `r` moves, because since they change the direction they need to be actually performed to have a consistent state;

3. Finally, if the error occurred while a plan was executing (99% of the times), it computes the new route to resume and finish the requested planning and goes to finish it.

#### Updating the waiterwalker.kb.pl file

Last but not least, we need to update the WaiterWalker to handle the information the Walker sends back with the error message. To do so, we added to its KB the correspondences between (obstacle ID, CurrDirection)-(ActualX, ActualY). Then, when it receives the error, it consults the KB and, if it finds a solution, it forwards it to the Walker, otherwise it accepts that there is an unresolvable error.

## The Manager Interface

The Manager Interface displays in a single view all the information regarding the tearoom state. To handle the updates coming from the Server, the javascript script handles the connection with the Server by using, like we did for the Client Interface in the previous SPRINT:

- a **SocketJS**, that connects to the Server's URL `'/it-unibo-iss'`;
- A **STOMP Client** that connects through the SocketJS and, once connected, subscribes to the `'topic/manager'` mapping to receive the updates.

Every time the manager receives a message from the Server, it updates the view accordingly by means of the javascript function `handleUpdateMessage()`.

## Refactoring of the Server

### State of the tearoom for the Manager

As previously stated in the SPRINT 3 document we are using the Spring Boot framework with an MVC Pattern.

What we introduce in this SPRINT is the usage of Services (`@Service` annotation), which are Components used to write business logic in a different layer, separated from `@Controller` class file. Then, each controller selects what services it will need to use by defining a constructor signature with the services it will need and then storing their reference inside its class.

What was all in the Client Controller is now split in four different services, one for each context:

- **tearoomService;**
- **barmanService;**
- **smartbellService;**
- **walkerService.**

Each service creates a connection with the Actors using a Coap Client to listen for updates from the waiter, walker, barman and smartbell and to eventually send messages to them.

When an update is sent by an Actor the service will extract the relevant information from the JSON and process it to set the fields in the following **State Singletons**:

- **WaiterState.java**

◦ <code>int freeTables;</code>	◦ <code>int positionX;</code>
◦ <code>int deployedToTable;</code>	◦ <code>int positionY;</code>
◦ <code>int teasDelivered;</code>	◦ <code>String currentTask;</code>
◦ <code>int dirtyTables;</code>	◦ <code>String</code>
◦ <code>int deployedToExit;</code>	<code>currentMovement;</code>
◦ <code>int earnings;</code>	

- **SmartbellState.java**

- `int clientsProcessed;`
- `int clientsAdmitted;`
- `int clientsWaiting;`
- `int clientsWaitedLong;`
- `String currentTask;`

- **BarmanState.java**

- `int ordersReceived;`
- `int teasPreared;`
- `int teasReady;`
- `String currentTask;`

Since we have put the business logic inside the Services, they don't have direct access to a `@SendTo` annotation or similar. Because of this, we need to assign to each Service a `SimpMessagingTemplate` variable which will handle the sending action.


## Interaction with the Client

The Services also need to send some required information to a specific Client. This interaction needs the Service to recognize what Client it needs to send a message to. To do this, we:

- Defined a custom handshake handler, so to assign a unique UUID to the connected client as its Principal;
- Add the security `@Configuration` and `@EnableWebSecurity` in a `SecureConfig` class, which inherits the `WebSecurityConfigurerAdapter` class;
- Setup the Websocket configuration to use our special handshake in the `WebSocketConfig` class and register the `/user` mapping, which is used to send messages in a non-broadcast way.

Once done this, we can retrieve the unique UUID by reading the incoming messages header and with that we can use the `SimpMessagingTemplate` method `ConvertAndSendToUser` to specify which user we want to send it to.

Since we also want to send messages asynchronously with respect to the request of the Client (i.e. when the drink is served or when a Client can enter the room), we need to store



the ClientID-UUID correspondence in a map. This map is kept by the WaiterService and used in the aforementioned occasions.

## ClientRequest

To improve the clarity of our code we have changed the ClientRequest's attributes from generic payloads to better named attributes:

- String name;
- String id;
- String table;
- String payment;
- String order;
- String clientid;
- String deployFrom;
- String deployTo;

## ServerReply

Similarly to Client Request, in ServerReply the attributes have been redefined.

- String redir;
- String clientid;
- String table;
- String waitTime;
- String result;

## Test

We run tests working with an exchange of messages to both check the Walker position and the correctness of the Waiter's behavior.