Politenico di Milano

Middleware Technologies for Distributed Systems

Project Report

# A Simple Smart Home

*Author:*
Fabio Losavio
Matteo Velati
Niccolò Zangrando

*Supervisor:*
Dr. Luca Mottola

April 11, 2021

## 1 Introduction

In a smart home, a control panel provides a user interface to coordinate the operation of HVAC systems based on environmental conditions gathered through sensors. Users input to the control panel their preferences, the desired room temperature and humidity, and the control panel returns to the user information on the instantaneous energy consumption over the Internet. The application is built using *Akka actor model* where one or more clients connects to a central server which is in charge of managing all the created rooms.

The project aims to implement a control panel to manage temperature and humidity of multiple rooms acting on a smart HVAC system. The user can connect, from any client, to a central server. There the user can:

- Create a new room
- Request to the server the list of all the registered rooms
- Request to a room the current data
- Set the target temperature and humidity of a room
- Delete an existing room

Communication between different parts of the application is carried out using different type of messages according to the operation.

## 2 Architecture

The application is built using the Akka actor model, all the actors communicate exchanging messages and can be distributed over the on different hosts over the network using the **Akka cluster functionalities**. As shown in Figure 2, a variable number of *clients*, which can be started by the users from any node on the network, will connect to the *central server* in order to manage and work on some parameters inside the *room servers* and to retrieve data produced by some sensors placed inside the room.

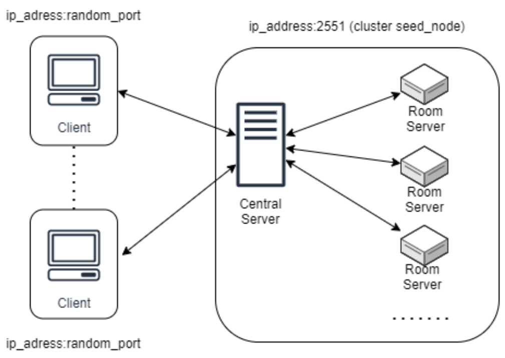


Figure 1: Application structure.

## 2.1 Client

Clients are Akka actors composed by two Java classes: a `ClientMain` and a `ClientActor`. Running the ClientMain, an actor will actually be instantiated and the application will allow the user to insert some commands. For each of the available operations, the ClientMain will trigger a self message to the ClientActor which will then forwards the same created message to the central server. In the following code section (Listing 1) it is possible to see all the different types of messages that the client can receive.

Listing 1: ClientActor createReceive function

```java
public class ClientActor extends AbstractActor{
[...]
    @Override
    public AbstractActor.Receive createReceive() {
        return receiveBuilder().
            match(CreateRoomMsg.class, this::onCreateRoomMsg).
            match(GetRoomsMsg.class, this::onGetRoomsMsg).
            match(GetDataMsg.class, this::onGetDataMsg).
            match(SetDataMsg.class, this::onSetDataMsg).
            match(CrashMsg.class, this::onCrashMsg).
            match(Response.class, this::onResponse).
            match(DeleteRoomMsg.class, this::onDeleteRoomMsg).
            build();
    }
[...]
}
```

## 2.2 Central server

The Central server is an Akka actor set up to be a **cluster seed-node** assigning it a fixed ip address and port so that every new actor can connect to it to join the cluster. To start up the central server, the `ServerMain` must be started on one machine, this will just instantiate the actor and show all the outputs of the server printed whenever the actor performs some actions. All the work will be done by the `CentralServerActor`, it will receive messages from all clients, the createReceive function is almost identical to the client one (Listing 1), and send to the correct room the actual request.

The central server also acts as a **supervisor** for all the room actors so that if a room crashes, it will manage the process to turn it back up keeping all the old data still valid and available (Listing 2).

Listing 2: Central server supervisor strategy

```java
// implement a resume strategy after a crash to keep all
// the inserted settings in the room
private final static SupervisorStrategy strategy =
    new OneForOneStrategy(10, Duration.ofMinutes(1),
        DeciderBuilder.match(ResumeException.class, e ->
        SupervisorStrategy.resume()).build());
```

According to the specific message received by one of the connected clients, the server can perform some actions. Let's now analyze them more in depth.

**CreateRoomMsg** Upon the reception of a `CreateRoomMsg`, the central server will instantiate a new `RoomServerActor`, with the name specified by the user inside the message, and will add its name and actor reference in a list alongside all the other previously created rooms. This allows any client to connect to the new room by just knowing its name.

**DeleteRoomMsg** When the server receive a `DeleteRoomMsg`, it will search in the list of all the ActorRef if a room with the given name already exists. If that room is found, the server will first of all stop that actor and then will remove its reference from the list.

**GetRoomsMsg** In order to know what rooms are already available, the user can request to the central server a list of all the available rooms by sending a `GetRoomsMsg`, with this message, the server will get the names of all the already created room actors and send them back to the user.

**GetDataMsg** Once the user knows the name of the actor he wants to contact, he can send a `GetDataMsg` to retrieve all the set up data for that room. By default the central server will return to the user the current temperature and current humidity of that room. If the room has the HVAC active, which can be activated by setting up a target temperature or humidity, the user will also receive those target values and the instantaneous energy consumption of the system.

**SetDataMsg** With this message, every client can decide to set up the target temperature and humidity of a specific room. The user can decide to either set only one of them up or both. When the server receive this message, it will get the ActorRef from the list of all the rooms and send it a message.

**CrashMsg** The last message that the server can receive is the `CrashMsg`. This is the only message that is not triggered by the user. This message gets triggered randomly to simulate the crash of a room actor. This will trigger the supervisor strategy on the room which is crashing to bring it back up without losing the current state.

## 2.3 Room server

The Room server actor is another Akka actor instantiated by the central server whenever a client requests it. Each room is identified by a name, which has to be unique, and stores the information retrieved by the sensor. When the room is started for the first time, the sensors will read the room temperature and humidity of the room. Initially the target values are not set (set up to -100 in the code) and the HVAC is off so the energy consumption is set to 0. If a target value is set, the HVAC will turn on. Every 10 seconds, the system is able to increase or decrease the temperature by $0.1°C$ and change the humidity by 0.5% consuming an energy of $U = P_{HVAC} * dt$ for each working part of the systems.

In our example we set the HVAC power to 1000 Watt and we considered a dt of 10 seconds so the maximum instantaneous energy consumption will be of 20000 Joule.

The room server actor will receive messages from the central server, with the client as a sender, will create the right response and send it back directly to the client.

## 3 Testing & conclusions

In order to test the application, the only parameters that have to be changed are the IP addresses of the machine which will run the system. For what concerns the server, the user has to insert its IP address in the `server.conf` file (as shown in listing 3). Those configurations will assign the correct IP address to the server and will set it up to be the seed-node of the cluster which will grant access to the cluster to any other node. Now the user can start the `ServerMain` to run the server.

Listing 3: IP settings in the server.conf file

```
remote.artery {
    canonical {
      hostname = "SERVER_IP_ADDRESS"
      port = 2551
    }
  }

  cluster {
    seed-nodes = [
      "akka://System@SERVER_IP_ADDRESS:2551"
     ]
```

To set up the client, an additional step is required. The user has to change the `client.conf` file inserting both its and the server's IP addresses and also insert the server IP in the `ClientActor` class to correctly retrieve the server's reference (listings 4 and 5).

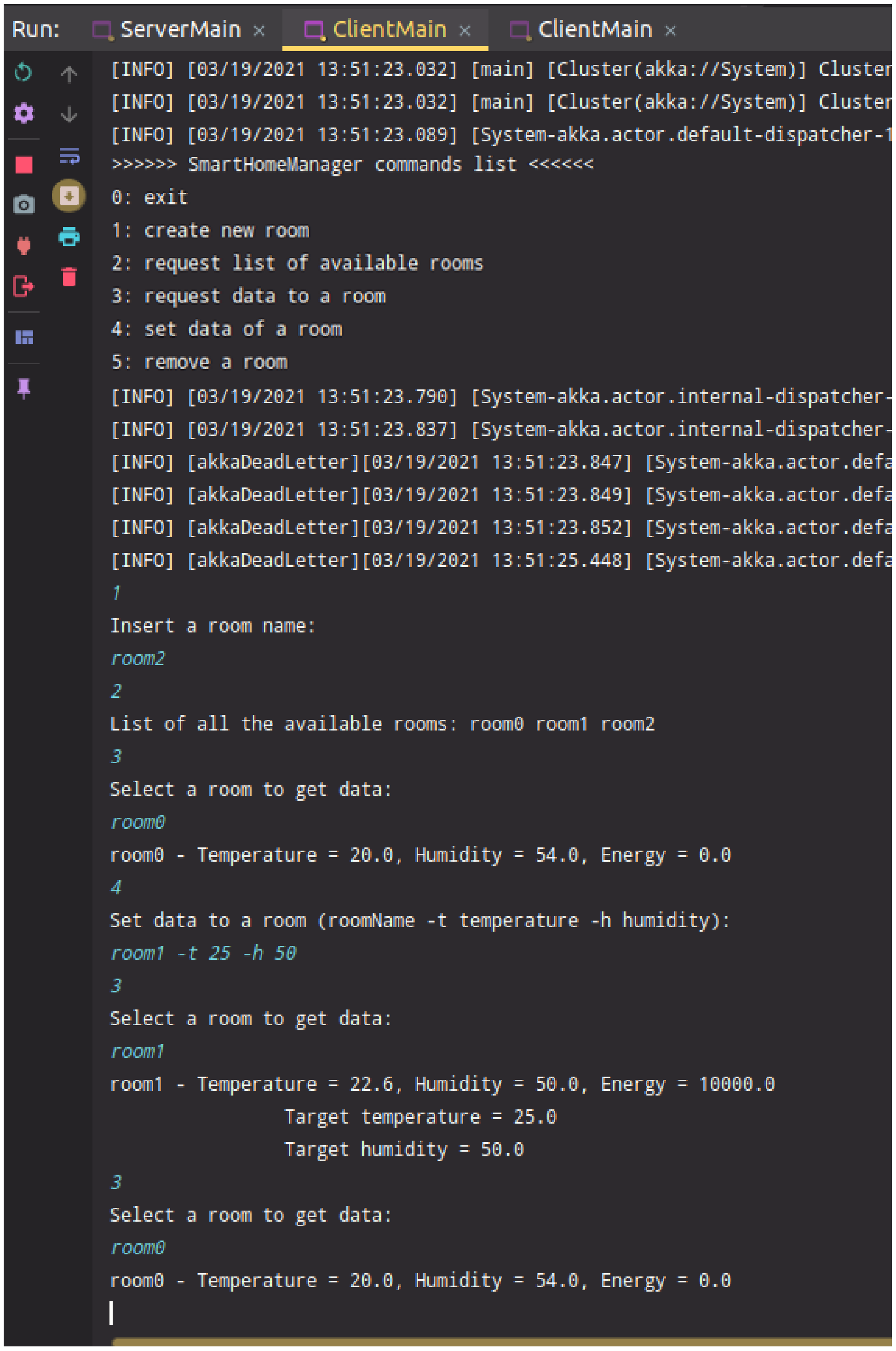
Listing 4: IP settings in the client.conf file

```
remote.artery {
    canonical {
      hostname = "CLIENT_IP_ADDRESS"
      port = 2551
    }
  }

  cluster {
    seed-nodes = [
      "akka://System@SERVER_IP_ADDRESS:2551"
     ]
```

Listing 5: IP settings in the ClientActor class

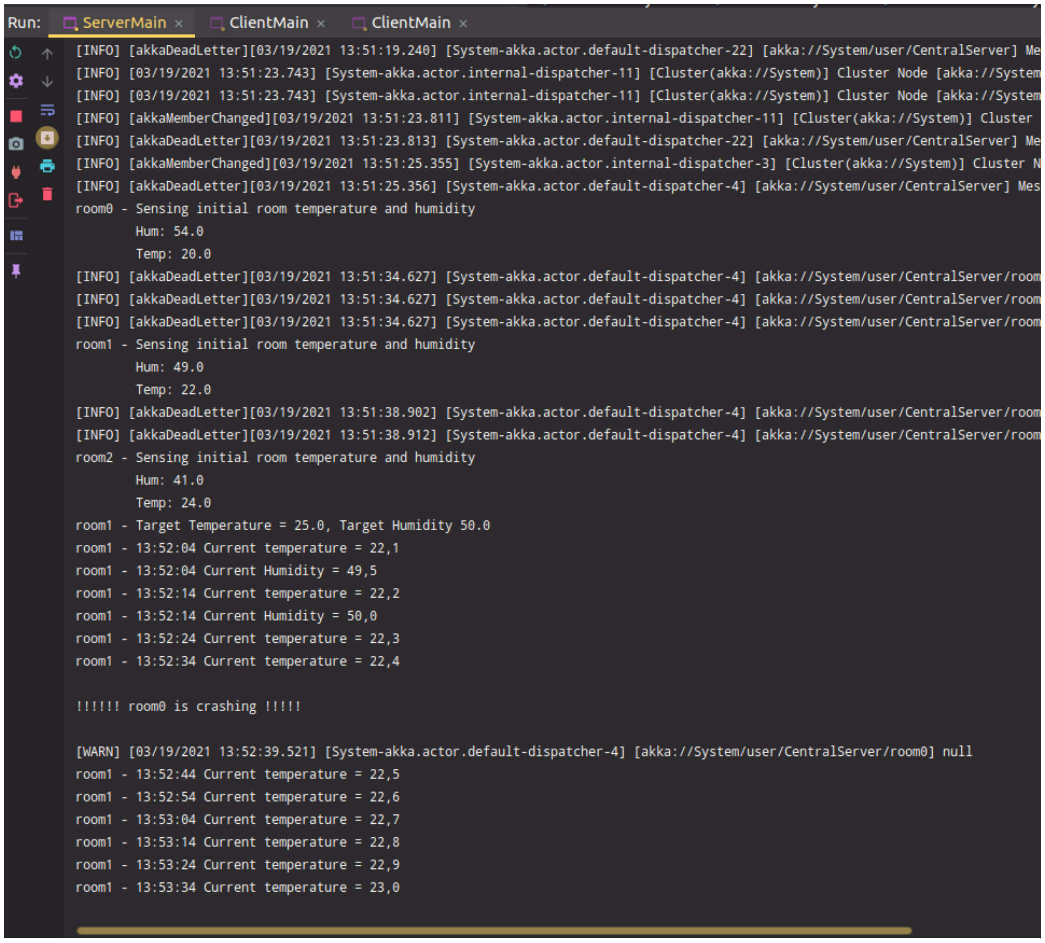
```java
public class ClientActor extends AbstractActor{

    final private ActorSelection server = getContext()
        .actorSelection(
        "akka://System@192.168.43.129:2551/user/CentralServer");

    [...]
}
```

To test if everything was working correctly, we tested the application first locally, using multiple processes on the same machine, and then in a distributed environment connecting multiple machines in the same network. In both scenarios everything worked as expected allowing every client in the cluster to correctly see all the changes issued by any of the other clients in the network. Thanks to the supervisor strategy, room crashes are totally hidden to clients and can only be noticed by looking at the central server console. Here there are some pictures of a sample test.



```
[INFO] [03/19/2021 13:51:23.032] [main] [Cluster(akka://System)] Cluster
[INFO] [03/19/2021 13:51:23.032] [main] [Cluster(akka://System)] Cluster
[INFO] [03/19/2021 13:51:23.089] [System-akka.actor.default-dispatcher-1
>>>>>> SmartHomeManager commands list <<<<<<
0: exit
1: create new room
2: request list of available rooms
3: request data to a room
4: set data of a room
5: remove a room
[INFO] [03/19/2021 13:51:23.790] [System-akka.actor.internal-dispatcher-
[INFO] [03/19/2021 13:51:23.837] [System-akka.actor.internal-dispatcher-
[INFO] [akkaDeadLetter][03/19/2021 13:51:23.847] [System-akka.actor.defa
[INFO] [akkaDeadLetter][03/19/2021 13:51:23.849] [System-akka.actor.defa
[INFO] [akkaDeadLetter][03/19/2021 13:51:23.852] [System-akka.actor.defa
[INFO] [akkaDeadLetter][03/19/2021 13:51:25.448] [System-akka.actor.defa
1
Insert a room name:
room2
2
List of all the available rooms: room0 room1 room2
3
Select a room to get data:
room0
room0 - Temperature = 20.0, Humidity = 54.0, Energy = 0.0
4
Set data to a room (roomName -t temperature -h humidity):
room1 -t 25 -h 50
3
Select a room to get data:
room1
room1 - Temperature = 22.6, Humidity = 50.0, Energy = 10000.0
                Target temperature = 25.0
                Target humidity = 50.0
3
Select a room to get data:
room0
room0 - Temperature = 20.0, Humidity = 54.0, Energy = 0.0
```

Figure 2: Sample of execution on a client.



```
[INFO] [akkaDeadLetter][03/19/2021 13:51:19.240] [System-akka.actor.default-dispatcher-22] [akka://System/user/CentralServer] Me
[INFO] [03/19/2021 13:51:23.743] [System-akka.actor.internal-dispatcher-11] [Cluster(akka://System)] Cluster Node [akka://System
[INFO] [03/19/2021 13:51:23.743] [System-akka.actor.internal-dispatcher-11] [Cluster(akka://System)] Cluster Node [akka://System
[INFO] [akkaMemberChanged][03/19/2021 13:51:23.811] [System-akka.actor.internal-dispatcher-11] [Cluster(akka://System)] Cluster
[INFO] [akkaDeadLetter][03/19/2021 13:51:23.813] [System-akka.actor.default-dispatcher-22] [akka://System/user/CentralServer] Me
[INFO] [akkaMemberChanged][03/19/2021 13:51:25.355] [System-akka.actor.internal-dispatcher-3] [Cluster(akka://System)] Cluster N
[INFO] [akkaDeadLetter][03/19/2021 13:51:25.356] [System-akka.actor.default-dispatcher-4] [akka://System/user/CentralServer] Mes
room0 - Sensing initial room temperature and humidity
        Hum: 54.0
        Temp: 20.0
[INFO] [akkaDeadLetter][03/19/2021 13:51:34.627] [System-akka.actor.default-dispatcher-4] [akka://System/user/CentralServer/room
[INFO] [akkaDeadLetter][03/19/2021 13:51:34.627] [System-akka.actor.default-dispatcher-4] [akka://System/user/CentralServer/room
[INFO] [akkaDeadLetter][03/19/2021 13:51:34.627] [System-akka.actor.default-dispatcher-4] [akka://System/user/CentralServer/room
room1 - Sensing initial room temperature and humidity
        Hum: 49.0
        Temp: 22.0
[INFO] [akkaDeadLetter][03/19/2021 13:51:38.902] [System-akka.actor.default-dispatcher-4] [akka://System/user/CentralServer/room
[INFO] [akkaDeadLetter][03/19/2021 13:51:38.912] [System-akka.actor.default-dispatcher-4] [akka://System/user/CentralServer/room
room2 - Sensing initial room temperature and humidity
        Hum: 41.0
        Temp: 24.0
room1 - Target Temperature = 25.0, Target Humidity 50.0
room1 - 13:52:04 Current temperature = 22,1
room1 - 13:52:04 Current Humidity = 49,5
room1 - 13:52:14 Current temperature = 22,2
room1 - 13:52:14 Current Humidity = 50,0
room1 - 13:52:24 Current temperature = 22,3
room1 - 13:52:34 Current temperature = 22,4

!!!!!! room0 is crashing !!!!!

[WARN] [03/19/2021 13:52:39.521] [System-akka.actor.default-dispatcher-4] [akka://System/user/CentralServer/room0] null
room1 - 13:52:44 Current temperature = 22,5
room1 - 13:52:54 Current temperature = 22,6
room1 - 13:53:04 Current temperature = 22,7
room1 - 13:53:14 Current temperature = 22,8
room1 - 13:53:24 Current temperature = 22,9
room1 - 13:53:34 Current temperature = 23,0
```

Figure 3: Sample of execution on the server.