

Documentation - Milestone 1

Domain Model Description

We will build an Internet Of Things (IoT) application running in a local area network, with several nodes communicating and passing each other data to create a connected infrastructure.

“Nodes” are basically IoT-devices equipped with Raspberry Pis, which will run our software. Each node will host an application offering services dependant on the underlying device, and will function as both client and server at the same time.

As each device is physically apart from all other devices, yet needs information from them to offer its full functionality (eg. a coffee machine would make coffee when the alarm clock rings), we can speak of a Distributed System.

The servers act as “listeners” and provide the entry point for all messages and/or notifications from the network, which are usually sent by the clients. Clients mainly serve as message senders. Using UDP, they broadcast the device-specific service offerings into the network using a predefined port, which all servers listening on this port can receive and analyze. Each service offering contains a unique code describing the service which is hosted on the device.

As already mentioned, the services offered by the application differ per node/device. It is important to note that the application itself never communicates with the network directly; all communication with other applications happens via server and client.

We mainly distinguish between two kinds of communication: messages from a human user (via a GUI), and messages from other IOT-devices.

As each device serves as both server and client, there is only one GUI for interacting with them. Human users will mainly be interested in interacting with the application, and for this the software offers a web-based GUI. At the moment, only the “alarmClock” service is implemented. Users can set and remove alarms; in the future, they will also be able to choose what actions other devices will take when an alarm goes off.

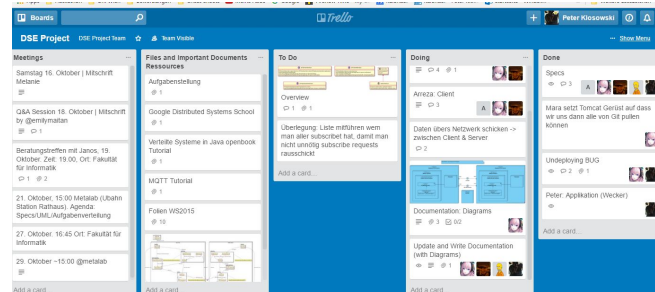
Messages between IoT-devices are beyond the user’s control and happen transparently in the background. Servers are permanently listening for service offerings which are regularly sent out by the clients. Since servers

are generic for all devices, they do not inherently know which offerings are needed, and need to ask the application whether it is interested.

Implementation Details

General

For easier organization, planning and tracking of the project we decided to maintain a Trello board where every information is stored and visible for every team member. It helped us to keep everybody up to date and to quickly see what isn't finished, yet.



To keep up with software updates concerning library dependencies, we decided to go with Maven. Maven allowed us to manage external java libraries for our software without pushing all of them to the main branch and got rid of the hassle to download each correct version for actual users of our program.

When designing our software, one of our baselines was writing clean code with focus on modularization and minimal dependencies (dependency injection). As a result, the codebase for server and client is shared across all devices; the one for application is not, but all applications offer the same interface which client and server can interact with.

Server

As we want our software to meet real-life requirements, we decided not to code a whole server by hand, and chose to implement some functions with the Apache Tomcat framework, which natively offers many functionalities we need (such as multithreading through HTTP-Servlets).

As already mentioned above, the server's main task is listening for incoming messages. This happens via the class `HTTPServerConnector`. Every GET-request is interpreted as a new subscription-request, while

each POST-request is treated as incoming message from a node the local application has subscribed to.

The main business logic of the server lies in the class `IOTServer`, which implements the interface `IOTServerInterface`. This class does the main work of analyzing incoming messages.

The Server-package also contains the package `UDPListener`. This is mainly for logical reasons - after all, listening for messages is the main purpose of the server - but the listener is actually started in the runner.

Client

The main function of all classes in the client-package is sending messages across the local network. Similarly to the server, the main business logic sits in the class `IOTClient`.

Currently, three types of such messages exist: service offerings, subscription requests and messages for subscribers.

Service offerings are short Strings known only to the application, and describe the service it offers. The class `UDPBroadcastService` packs this description into UDP datagrams and regularly (currently once per second) floods the network with them, using the broadcast-address 255.255.255.255 and port 29902.

Subscription requests are basically the “answer” to a service offering received from another device. The method `createSubscriptionRequest` (which is called by the server after asking the application if it is interested) in `IOTClientInterface` is responsible for creating this request.

Messages for subscribers are sent via the method `notifySubscribers` of the same interface. The format used for these messages is JSON.

Application

Every application that will be running on the Raspberry Pi consists of different methods. One of them allows the Server to forward incoming, and for this application important, notifications. The other two are information for the Server to get the current service description (ID i.e `AlarmClockService: ACS101`) and if another service is relevant enough to subscribe to.

These functions are defined in the Application Interface and every application derives from that. Right now there's just one implemented application, the `AlarmClockService`. It reports to the instance of the client

when it sounds an alarm. The user can configure alarms through a GUI, that includes setting, cancelling, stopping and removing them.

When an alarm is started, that means turned on to sound at a specific date in the future, a Task will be created and scheduled to be run in a new Thread at that time. This thread starts an instance of PiezoPlayer that plays a predefined melody with the connected Piezo element if it's being executed on a Raspberry Pi, otherwise just a console log will be printed. At that moment also a notification will be sent out through the client to every subscriber and the alarm will be turned off.

Other planned service implementation are a coffee maker, light bulb control, IOT button and a temperature listener. Further ideas for implementations were (door) lock control, power outlet control, camera control (movement, face recognition), music system control, air quality listener and voice control.

Program Execution Information

The Program was designed to be deployed and run on Raspberry Pis, therefore it's not guaranteed that every function is working on other platforms or hardware as well. Some functions are cut short or different like an alarm will be just printed to the console instead of playing a melody.

Used Hardware, Software and Libraries

What IDEs, Libraries, ... did we use with versions.

- Raspberry Pi 2
- Plantuml / Visual Paradigm 13.1
- Eclipse 4.6.1 / IntelliJ IDEA 2016.1.4
- Java 1.8.0_111
- Apache Tomcat 8.0.38
- Maven Dependencies:
 - GSON 2.7
 - Java Servlet API 3.1.0
 - Jersey Core Server 2.23.2 (Server & Client)
 - JUnit 4.12

- Pi4J 1.1

Additional comments and notes

Fun Fact: We decided to use port 29902 because after finding out that root user rights are necessary for port 1000 and already having a mario theme for the alarm clock it was just too clear for us to use a port number that is between other port numbers used by Nintendo themselves (29901 & 29920).