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.

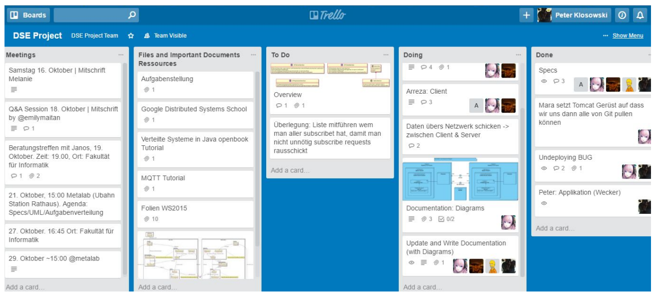
As each device serves as both server and client, there is only one GUI for interacting with them. Human users will mainly be interested in interacting with the application, and for this the software offers a web-based GUI. At the moment, only the “alarmClock” service is implemented. Users can set and 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.



Meetings
- Samstag 16. Oktober | Mitschrift Melanie
- Q&A Session 18. Oktober | Mitschrift by @emilymaitan
- Beratungstreffen mit Janos, 19. Oktober. Zeit: 19.00, Ort: Fakultät für Informatik
- 21. Oktober, 15:00 Metalab (Ubahn Station Rathaus). Agenda: Specs/UML/Aufgabenverteilung
- 27. Oktober. 16:45 Ort: Fakultät für Informatik
- 29. Oktober ~15:00 @metalab

Files and Important Documents Ressources
- Aufgabenstellung
- Google Distributed Systems School
- Verteilte Systeme in Java openbook Tutorial
- MQTT Tutorial
- Folien WS2015

To Do
- Overview
- Überlegung: Liste mitführen wem man aller subscribet hat, damit man nicht unnötig subscribe requests rausschickt

Doing
- Arreza: Client
- Daten übers Netzwerk schicken -> zwischen Client & Server
- Documentation: Diagrams
- Update and Write Documentation (with Diagrams)

Done
- Specs
- Mara setzt Tomcat Gerüst auf dass wir uns dann alle von Git pullen können
- Undeploying BUG
- Peter: Applikation (Wecker)

To keep up with changing library dependencies, we decided to go with Maven. Maven allows us to manage external java libraries for our software without pushing all of them on GitLab, and eliminates the hassle of having to download each correct version for actual users of our program.

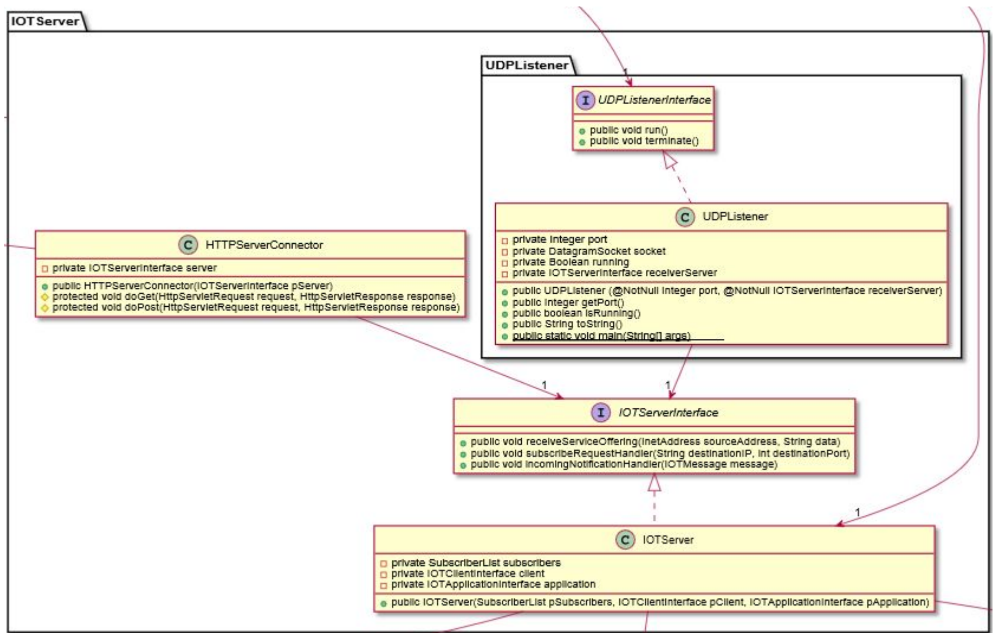
When designing our software, one of our main goals was writing clean code with focus on modularization and minimal dependencies (dependency injection). As a result, the codebase for server and client is shared across all devices; the one for application is not, but all applications offer the same interface which client and server can interact with. The only other files that need to be adapted for each node are web.xml and the IOTRunner.

Concerning the division of work, the number of commits on gitlab does in no way reflect the amount of work each team member invested. There is a total of six team meetings documented in Trello, each of which all team members attended. We spent many hours carefully planning, sketching and discussing design, structure and frameworks / deployment before assigning tasks to every team member. We eventually split up like this:

- Server: Aichinger Mara, Balaz Melanie
- Client: Fetahaj Arreza, later Aichinger Mara and Balaz Melanie
- Application: Klosowski Peter
- Android Application (future milestones): Eichinger Rene

For each task, we created a separate branch, as well as a developer branch for common features (interfaces) and merging everything together.

Server


IOTServer
UDPListener
UDPListenerInterface
public void run()
public void terminate()
UDPListener
private Integer port
private DatagramSocket socket
private Boolean running
private IOTServerInterface receiverServer
public UDPListener (@NotNull Integer port, @NotNull IOTServerInterface receiverServer)
public Integer getPort()
public boolean isRunning()
public String toString()
public static void main(String[] args)
HTTPServerConnector
private IOTServerInterface server
public HTTPServerConnector(IOTServerInterface pServer)
protected void doGet(HttpServletRequest request, HttpServletResponse response)
protected void doPost(HttpServletRequest request, HttpServletResponse response)
IOTServerInterface
public void receiveServiceOffering(InetAddress sourceAddress, String data)
public void subscribeRequestHandler(String destinationIP, int destinationPort)
public void incomingNotificationHandler(IOTMessage message)
IOTServer
private SubscriberList subscribers
private IOTClientInterface client
private IOTApplicationInterface application
public IOTServer(SubscriberList pSubscribers, IOTClientInterface pClient, IOTApplicationInterface pApplication)
1
1
1
1

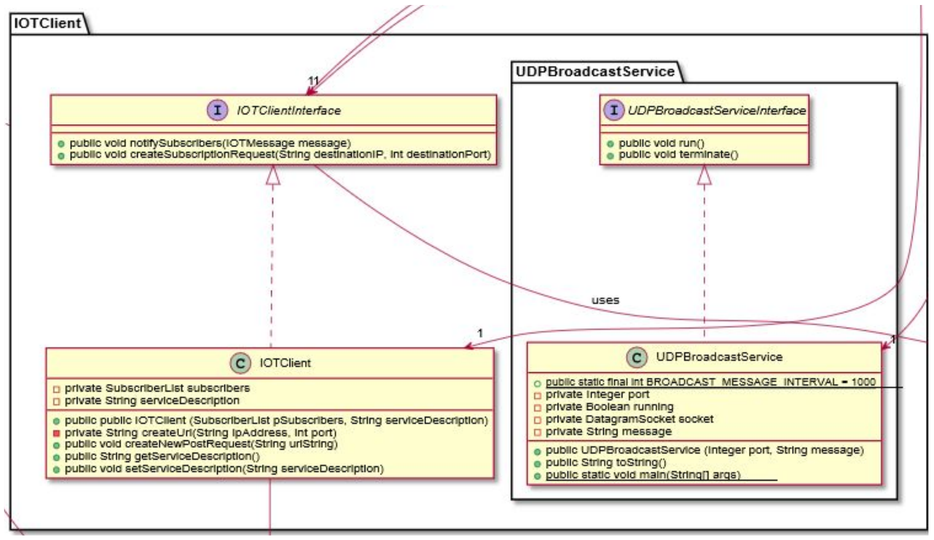
As we want our software to meet real-life requirements, we decided not to code a whole server by hand, and chose to implement some functions with the Apache Tomcat framework, which natively offers many functionalities we need (such as multithreading through HTTP-Servlets).

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


IOTClient
11
IOTClientInterface
public void notifySubscribers(IOTMessage message)
public void createSubscriptionRequest(String destinationIP, int destinationPort)
UDPBroadcastService
UDPBroadcastServiceInterface
public void run()
public void terminate()
uses
1
IOTClient
private SubscriberList subscribers
private String serviceDescription
public public IOTClient (SubscriberList pSubscribers, String serviceDescription)
private String createUrl(String ipAddress, int port)
public void createNewPostRequest(String urlString)
public String getServiceDescription()
public void setServiceDescription(String serviceDescription)
1
UDPBroadcastService
public static final int BROADCAST_MESSAGE_INTERVAL = 1000
private Integer port
private Boolean running
private DatagramSocket socket
private String message
public UDPBroadcastService (Integer port, String message)
public String toString()
public static void main(String[] args)

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.

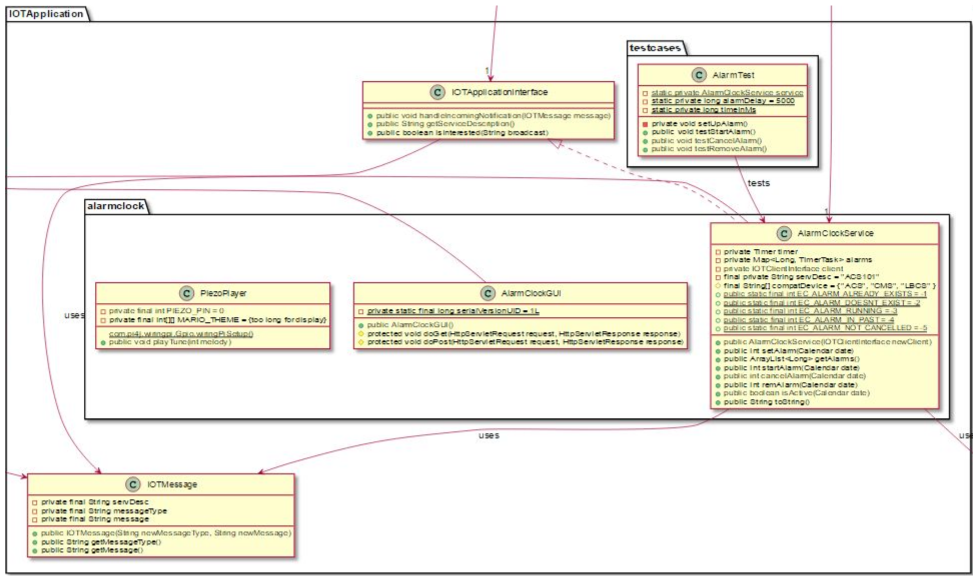
Currently, three types of such messages exist: service offerings, subscription requests and messages for subscribers.

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


IOTApplication
testcases
IOTApplicationInterface
public void handleIncomingNotification(IOTMessage message)
public String getServiceDescription()
public boolean isInterested(String broadcast)
AlarmTest
static private AlarmClockService service
static private long alarmDelay = 5000
static private long timeInMs
private void setUpAlarm()
public void testStartAlarm()
public void testCancelAlarm()
public void testRemoveAlarm()
tests
1
1
alarmclock
uses
PiezoPlayer
private final int PIEZO_PIN = 0
private final int[][] MARIO_THEME = {too long for display}
com.pi4j.wiringpi.Gpio.wiringPiSetup()
public void playTune(int melody)
AlarmClockGUI
private static final long serialVersionUID = 1L
public AlarmClockGUI()
protected void doGet(HttpServletRequest request, HttpServletResponse response)
protected void doPost(HttpServletRequest request, HttpServletResponse response)
AlarmClockService
private Timer timer
private Map<Long, TimerTask> alarms
private IOTClientInterface client
final private String servDesc = "ACS101"
final String[] compatDevice = {"ACS", "CMS", "LBCS" }
public static final int EC_ALARM_ALREADY_EXISTS = -1
public static final int EC_ALARM_DOESNT_EXIST = -2
public static final int EC_ALARM_RUNNING = -3
public static final int EC_ALARM_IN_PAST = -4
public static final int EC_ALARM_NOT_CANCELLED = -5
public AlarmClockService(IOTClientInterface newClient)
public int setAlarm(Calendar date)
public ArrayList<Long> getAlarms()
public int startAlarm(Calendar date)
public int cancelAlarm(Calendar date)
public int remAlarm(Calendar date)
public boolean isActive(Calendar date)
public String toString()
uses
us
IOTMessage
private final String servDesc
private final String messageType
private final String message
public IOTMessage(String newMessageType, String newMessage)
public String getMessageType()
public String getMessage()

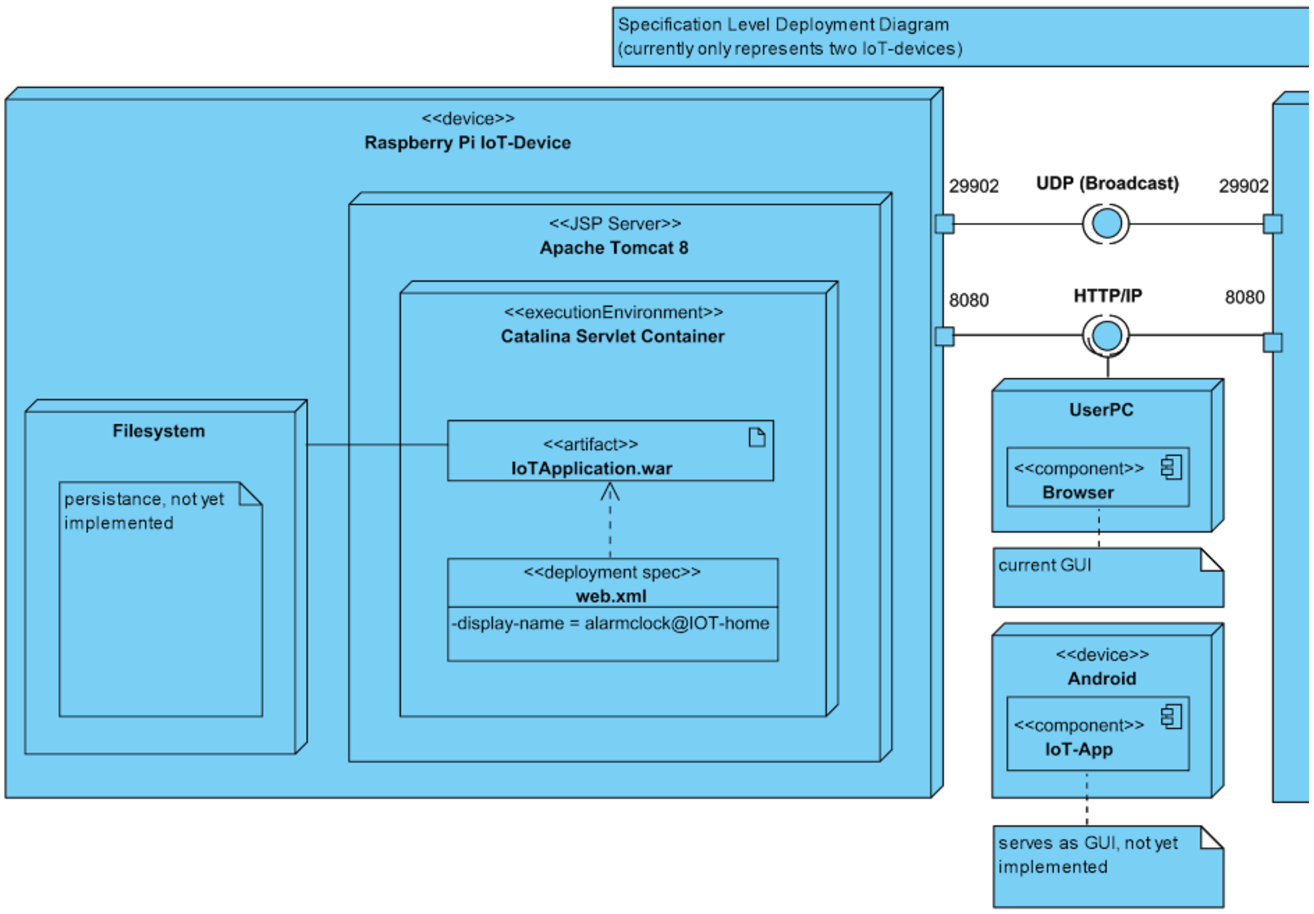
Every application that will be running on the Raspberry Pi consists of different methods. One of them allows the Server to forward incoming, and for this application important, notifications. The other two are information for the Server to get the current service description (ID i.e AlarmClockService: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

Physical Deployment

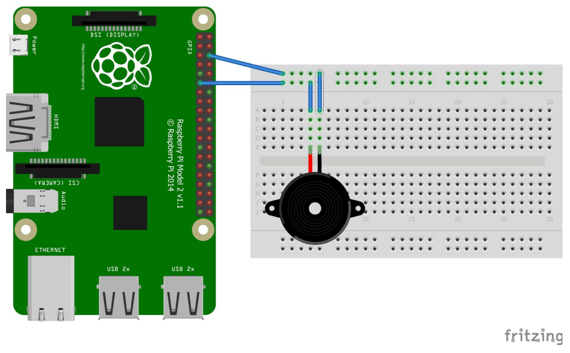


The above specification-level deployment diagram aims to give an overview how our software maps to the hardware. (Note that the cropped of the diagram only contains another Raspberry Pi IoT node with only differs from the visible one by the display-name in web.xml. The full diagram can be found on GitHub.)

The <<device>> Raspberry Pi IoT-Device represents one node in our network. Each IoT Device runs its own instance of Tomcat (and hence also our software), which as per default listens for HTTP GET and POST requests (PUT and DELETE will possibly be implemented in future milestones) on port 8080. The only other port in use by our software is port 29902, via which all UDP service discovery messages are handled.
It is important to note that, in order for our software to work correctly, all devices need to be connected to the same local network.

Program Execution Information

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



To run the program, it first needs to be exported as a war file from inside your IDE. It was tested and is compatible with Tomcat 8. After the deployment onto the server application (in a testing environment or a Raspberry Pi), you will be greeted by a webpage - given you visit the correct link on localhost (i. e. localhost:8080/g2016w_dse_0401/). There is a link mapping to the AlarmClockService GUI, where you can add new alarms.

Used Hardware, Software and Libraries

What IDEs, Libraries and Hardware did we use with the version numbers:

- 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).