CHALMERS

Infotainment and mobile devices

Connecting mobile devices with a vehicle's infotainment system using web technology

Bachelor of Science Thesis

DANIEL KVIST
PATRIK THITUSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden, 2013

ABSTRACT

In the automotive industry the infotainment system's today are locked down and controlled by the manufacturers with proprietary systems. This leads to lack of user freedom and long application development times. This report investigates ways of communicating between services on the Internet, mobile devices and a computer system in a motor vehicle using free and open software. During the execution of this project a communication platform based on MQTT have been developed together with a sample set of Android applications that demonstrate the concept and functionality of the system. The system allows a device to push web applications into a simulated vehicle environment which handles the display and communication of the installed application. If a system like the one that has been developed during this project would be implemented in the next generation vehicles, it is clear that the personalisation of content and data will take the next step from smartphones into cars. It will also allow the time it takes to develop and deploy infotainment applications to reduce significantly.

Java is to JavaScript what Car is to Carpet.

— Chris Heilmann

ACKNOWLEDGMENTS

When we started on this project, we really did not know where we would land. Using web and mobile device technology inside a car with sensors, head units, screens and new protocols like MQTT actually made us a bit confused. Where to start? Where to end? How to delimit?

During the project we have slowly but steady grasped how all these technologies could work together and we are happy with the end result. The path to the end has not been straight however, not ever. Many failures, misconceptions and misunderstandings have somehow led us to a successful project though, but we could not have done it without the help of the people around us, at home and at work.

A special thank you to Jens Cederhage and Oskar Hagberg at Cybercom Group that gave us a lot of freedom, but also great pointers and tips and ideas on what could be useful and not. They also showed patience and trust in that we would be able to finish this project. We could not have done it without you.

Another thank you to Joachim von Hacht at Chalmers University of Technology for your early input on pro's and con's with the technologies we would be investigating. We thank you again for your efforts in helping us structure, proof read and produce this report.

Gothenburg, 2013-05-26
Patrik Thituson, Daniel Kvist

CONTENTS

TERMINOLOGY

- Broker

  The central server in an MQTT communication system.

- Head unit

  Placed in the center stack of a car, usually contains a screen with information about the car infotainment or climate system.

- Infotainment

  Short word in the automotive industry for Information and Entertainment. Describes music/video/entertainment in vehicles.

- OEM

  Original Equipment Manufacturer, builds equipment such as cars, mobile devices etc.

- MQTT

  Message Queue Telemetry Transport is a communication protocol commonly used for machine to machine communication.

- Publisher

  A client in an MQTT network that publishes a message to the network.

- Subscriber

  A client in an MQTT network that subscribes to a topic to receive messages.

- Topic

  A common channel, topic, subject that MQTT clients can subscribe and/or publish to.

1 INTRODUCTION

1.1 BACKGROUND

Cybercom is an IT consulting company that assists companies and organisations to make the most of the opportunities in a connected world. The Gothenburg department of Cybercom focuses on embedded systems and wireless communications. There have been an increasing need and demand from customers and employees to explore new ways to use these ever evolving technologies in “infotainment” systems that can be found in cars for example.

In todays’ infotainment systems it takes a long time to implement new features. A product planning cycle of several years is common. When the systems are finally delivered they are locked down and use proprietary solutions. This means that the technology and services that are offered in the infotainment systems does not integrate well, if at all, with the Internet services that end users are using today.

If it was possible to simplify development of new infotainment applications for the automobile industry it would be a benefit to the user. This report will look into some of the possibilities and issues that arise when trying to open up the vehicle’s infotainment system to web technology.

1.2 USE CASE

1.2.1 BACKGROUND

A fleet of vehicles often have different drivers in different vehicles. It can be a rental car company, carpool, delivery firm, taxi etc. The user of the vehicles have a need for bringing their own custom content into the car. This content can consist of mail, music, movies, phone contacts and such. Enabling such a content sharing system can be achieved by pushing personalised applications from a mobile device into the car.

1.2.2 STORYLINE

Emma and Simon are going on a trip and have rented a car from Hertz. Emma takes the drivers seat and connects her music application on her phone with the car. She can then install another child application by pushing it from the device into the head unit of the car. The installed application will show the playlist of the music application.

Emma can control the application through the car’s input system, i.e buttons on the steering wheel, and select a song. Simon that sits in the passenger seat has another application that also can control the application in the head unit. He really wants to hear another song and adds it to the playlist after connecting his device. Since Emma and Simon can share a combined playlist they will have a pleasant trip without arguing.

1.3 PURPOSE AND GOAL

1.3.1 PURPOSE

The purpose of this project has been to investigate the possibility of using web technologies to communicate between services on the internet, mobile devices and the computer system in a car.

1.3.2 GOAL

The goal was to create a system of applications that demonstrate the communication between a mobile application and the computer system in a car where the mobile application can push a web application into the car. The concept will allow for further research and expansion, which will allow drivers and passengers in a car to bring applications with them into a new environment where the car is seen as an i/o interface for mobile devices and their services.

1.4 PROBLEM DESCRIPTION

To be able to fulfill the goal and purpose of the project, the following questions needed to be answered.

- What communication architecture pattern is suitable to use?
- What communication protocols should be used?
- How should the application be delivered to the car?
- How can the car and device send data to each other?

1.5 DELIMITATIONS

The project did not aim to fulfill any legal requirements that might be in place today. Neither was there any focus on the connection between the car and the screen in the car where the application would be presented.

The security implications of a system like the one that will be developed are many and severe. There will not be any focus on security in the concept applications.

It will be assumed that the physical communication channel is already available in the vehicle. In this project, WiFi will be assumed.

2 METHOD

To find out what it would take to develop a communication system like the one described in the introduction, a prototype communication system was created. Since it would be too expensive to develop in a real environment with a car, the car's system was simulated. To develop such a prototype, it was necessary to study literature about mobile operating systems, web technology and communication protocols.

3 TECHNOLOGY

3.1 CAR COMPUTER SYSTEM

In a car between the front seats the “head unit” is placed which can be seen in figure 1. The head unit contains a display, climate control, music control and more. It is becoming increasingly common that the display in the head unit also is touch enabled. In today’s current vehicles, the applications running in the head unit are all controlled by the manufacturer.

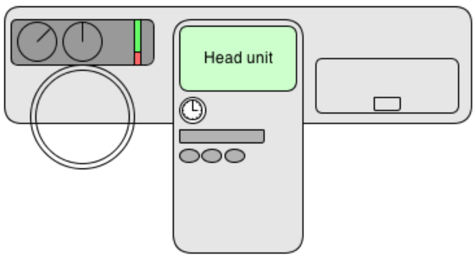

Figure 1: The head unit in a modern vehicle.

There are many systems in a car that communicate and which may contain interesting data for a third party developer. The sensor communication from the Electronical Control Unit’s (ECU) is handled by the Controlled Area Network (CAN) bus and a Local Interconnect Network (LIN). In traditional systems, the head unit has been displaying information from the ECU’s directly. This is shown in figure 2.

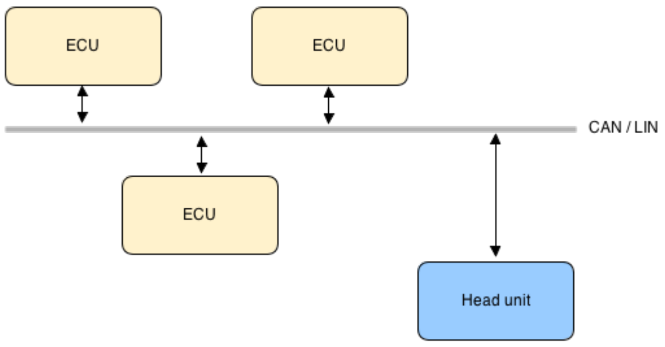

Figure 2: Simplified overview of a car computer system

3.2 MQTT

Message Queue Telemetry Transport (MQTT) is a message passing protocol that is lightweight and efficient. It uses a publish/subscribe message passing model which makes it easy to distribute data to any number of clients through a topic.

To use MQTT a message broker is needed. The broker acts as a server which receives published messages from all publishers. It also keeps track of all subscribers and forwards published messages to subscribers of the topic that a message was published to.

3.2.1 MQTT FLOW DIAGRAM

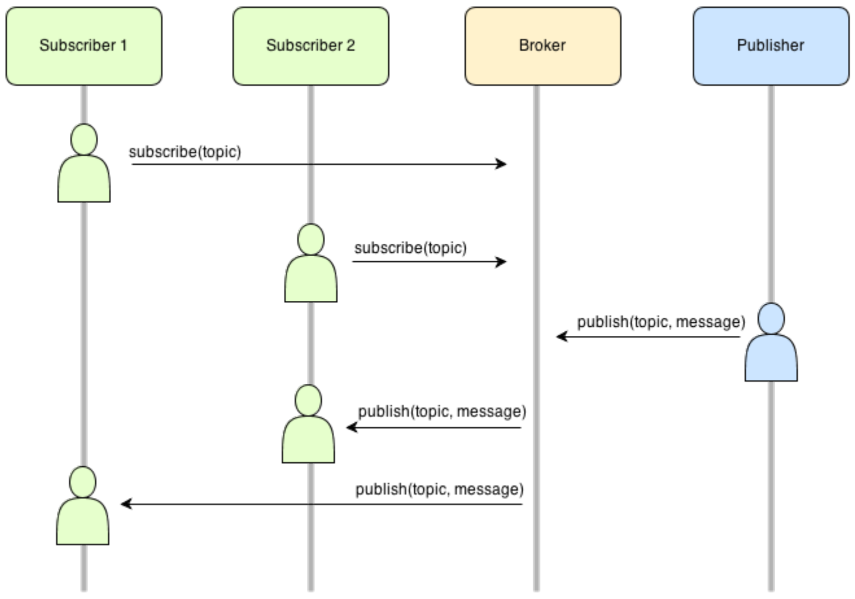

Figure 3: Communication between publishers and subscribers

Figure 3 shows the basic principle behind the MQTT protocol. Each MQTT client that subscribes to a topic and becomes a subscriber. The broker keeps track of all subscribers so that when a publisher publishes a message on a given topic, all of the subscribers of this topic will receive the message.

The MQTT message has a very simple structure. It consists of a topic and a payload. The payload can be up to 256MB while the message header consists of only two bytes [1]. This makes the protocol easy to use and understand in addition to being lightweight and flexible. A message can have three different quality of service settings which allows the protocol to prioritise and save messages for later delivery. It is also possible to retain messages at the broker which gives MQTT the possibility to store persistent data.

3.2.2 MQTT AND HTTP COMPARISON

HTTP which is used on the Internet today is a well known protocol that uses a different communication pattern. The table below compares MQTT with HTTP: [2]

	MQTT	HTTP
Design orientation	Data centric	Document centric
Pattern	Publish/subscribe	Request/response
Complexity	Simple	More complex
Message size	Small	Large
Service levels	Three qos	Messages same level of service
Extra libraries	Usually very lightweight	Depends on the application
Data distribution	Supports 1-1, and 1-n	1-1 only

3.3 MOSQUITTO

Mosquitto is a widely used open source MQTT broker written in Python. It implements the full MQTT 3.1 specification and it has been used since 2009 in various projects. It supports persistence which allows it to store data through power cycles. Cybercom has worked with this broker in the past and it would be a stable choice of broker to use in a “real” implementation of an MQTT system. [3]

3.4 NODE.JS

Node.js is a framework for writing server-side applications using JavaScript. It runs in a single thread and uses an event-driven, asynchronous I/O model. This makes it lightweight and scalable. The flow of execution in an event-driven programming style is determined by events which means that they invoke a callback function when an event has occurred. [4]

Since Node.js is using JavaScript which is widely used for the client side in the web-technologies it would be a natural extension to use it for the server-side as well and therefore an option for our broker implementation.

3.5 HTML5 AND RELATED TECHNOLOGIES

There are many different definitions of what HTML5 means. Some claim it is just the new HTML specification and nothing else, others wish to include CSS3 and the new Document Object Model (DOM) in the specification as well. In this report HTML5 and its related technologies will simply be referred to as HTML5. Where it is required to be more specific; CSS3, JavaScript and the DOM will be referred to as such. [5]

HTML5 also offers access for each domain to a local storage where the application can store data as key value pairs. This can be utilized by applications to maintain a persistent state which is the same all across the Internet and in other HTML5 applications. It also offers WebSockets. A WebSocket provides full duplex communication over a Transmission Control Protocol (TCP) connection [6].

Together with MQTT and Node.js, HTML5 can offer a platform of communication between services on the internet, mobile devices and a central processing unit in a motor vehicle. If it is possible to bring HTML5 web applications into a car, it would greatly simplify the process of developing new infotainment system applications.

3.6 ANDROID

Android is an open-source mobile Operating System (OS) created by Google and is today the largest smartphone OS in the world [7]. The open nature of the operating system makes it suitable for experimenting and testing new technology on.

3.7 NEAR FIELD COMMUNICATION (NFC)

NFC is a set of short-range wireless technologies that require a very short distance to initiate a connection. It is most commonly used by smartphones and other mobile devices. When two NFC chips come into close proximity of each other, they are activated and the connection that is established can be used to send data. NFC can also be found in tiny stickers and tags which can be used to connect to devices that can then use the data from the tag for some action.

3.8 JAVASCRIPT OBJECT NOTATION (JSON)

JSON is a lightweight data-interchange format that is language independent but uses conventions which are very similar to the ones found in C, Java, Python and other languages. It is fast, simple and readable. JSON provides a flexible, powerful and easy to understand approach to data transport.

4 WORKFLOW

4.1 INITIAL STAGE

The project was executed in three phases; research, analysis and implementation. During the research phase, different communication protocols and architectures were examined. It was then decided which technologies would be suitable to use as a base for the project. The criteria for choosing any technology was limited by availability, cost and time for implementation. The choices have been based on research and experiments with the different technologies to see which suited the development team's knowledge best. Note that the solutions for reaching the goals and purpose were specific for this project and there might be other technologies that suit other developers better.

4.2 PROJECT PLANNING AND EXECUTION STAGE

4.2.1 PROJECT METHODOLOGY

Since this project tested different technologies to reach the goal, it was necessary to plan the project execution with a flexible model. This is why it was decided to work agile by using Scrum during the implementation phase.

With the use of Scrum daily standup's and weekly Scrum planning meetings it was possible to quickly change approach if a technology could not fulfill the requirements.

5 REQUIREMENTS

5.1 NON-FUNCTIONAL REQUIREMENTS

Since the current infotainment systems that are used today are controlled by the Original Equipment Manufacturer's (OEM's), there was a request from Cybercom that the implementation should use open source and free software. The communication architecture had two main requirements that had to be fulfilled to be able to allow the applications to have a solid foundation;

- a client must be oblivious of any other clients in the system, it should only know about the broker
- a single server/broker must handle all message passing in the system

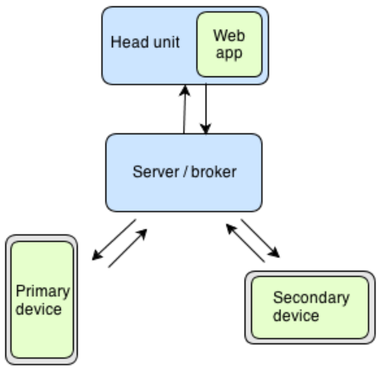

Figure 4: A simplified overview of the communication platform where the primary device is able to push web applications into the head unit and the secondary device can control the web app and the application running in the primary device.

5.2 FUNCTIONAL REQUIREMENTS

5.2.1 HEAD UNIT

The head unit displays a web application which is controlled by the primary device. It had seven main requirements that had to be fulfilled;

- subscribe to topics through the broker
- publish status messages to a device application's private topic
- receive installation packages and actions in published messages
- host and serve installed web applications
- show any requested web application on the screen
- forward any non-system messages to the web application
- allow the hosted web application to subscribe to topics and publish messages

5.2.2 PRIMARY DEVICE

The primary device contains the core application of the demonstration system. It is required for the system to work as it is responsible for installing and starting the web application that runs in the head unit. The primary device had seven requirements that had to be fulfilled;

- publish installation package and the actions start, stop, uninstall
- subscribe to topics through the broker
- host a playlist of tracks
- receive published tracks and add to the playlist
- receive the published actions ‘play’, ‘pause’, ‘next’ and ‘prev’
- play, pause, next, previous functionality for tracks
- publish the actions ‘play’, ‘pause’, ‘next’, ‘prev’

5.2.3 SECONDARY DEVICE

The secondary device is an add-on remote control to the demonstration system that can control playback and manage the playlist in the primary device. It is not required for the system to work. To demonstrate the features of the network the secondary device had to fulfill the following requirements;

- receive messages with actions that control the player controls
- receive messages with tracks that are added to the playlist
- search the Spotify Metadata API and display the results
- publish a message which adds the selected track to the playlist
- publish messages which control the playback of the track

5.2.4 WEB APPLICATION

The web application is installed by the primary device into the head unit. It can be started and stopped by the primary device, it also listens for control messages on other topics. It had four requirements that had to be fulfilled;

- receive actions from device app through the JavaScriptInterface in the head unit
- update the user interface as appropriate when an action is received
- allow the user to click control buttons to control the playlist
- send actions to device app through the JavaScriptInterface in the head unit

5.2.5 SENSOR GATEWAY

The sensor gateway publishes messages on a topic that the other applications are subscribing to. This allows it to control playback of the music. The sensor gateway application had three requirements which had to be fulfilled;

- publish a play action message
- publish a next action message
- publish a pause action message

6 DESIGN

6.1 DESIGN OF COMMUNICATION PLATFORM

6.1.1 PLATFORM OVERVIEW

The communication system and architecture could be split into several different parts;

- server/broker that receives and forwards published messages
- sensors from the vehicle's system
- gateway for the sensor data to the broker
- web application that can be installed and executed in the head unit
- head unit that hosts the web application
- primary device application that pushes the web application into the head unit and controls it
- secondary device application that can hook into the system and control playback and the playlist

These different parts would interact with each other through the server/broker which is the central component of the system.

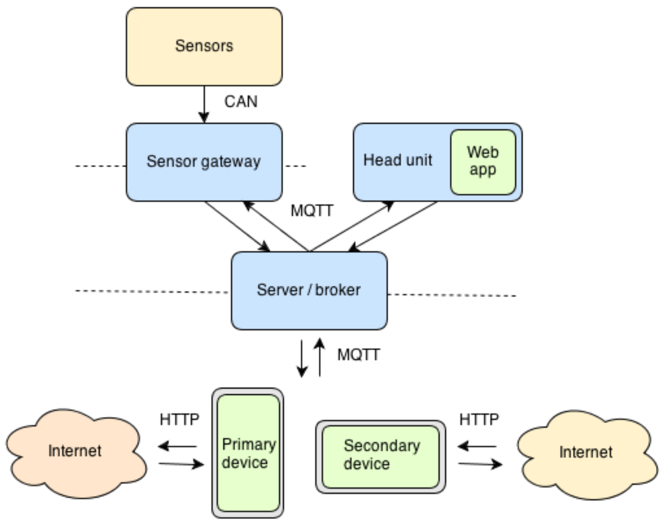

Figure 5: Architecture overview of the communication platform where the devices contain applications that communicate with the web application hosted in the head unit.

6.1.2 PROTOCOL AND DATA TRANSPORT

6.1.2.1 MQTT

MQTT and HTTP are two protocols that could be used in the implementation and MQTT was chosen. The biggest advantage of using MQTT over HTTP is that MQTT supports a greater number of possibilities when it comes to data distribution. The protocol is designed for communication between any number of clients. Its publish/-subscribe communication pattern is also superior to HTTP's request/response in a scenario where pushing data across a network is required and the number of clients may vary. The request/response pattern simply is not flexible enough for this type of implementation.

The use of MQTT also allows the system to add and remove MQTT clients at will and the other nodes doesn't have to know anything about it. This creates a very flexible communication architecture where application developers doesn't have to connect to specific clients, instead they just subscribe to topics which deliver messages.

6.1.2.2 DATA TRANSPORT

MQTT can use topic structures to send structured data. It is possible to publish a variety of different values, each on its own topic and topic naming conventions can vary extensively depending on what kind of data is being sent.

To allow for powerful applications to be implemented over the communication system it was decided not to use extensive topic structures. An application may need to send complex data structures and using topics to accomplish this would be inconvenient and messy. To send an object with four fields for example, four different messages would have to be sent and when adding a synchronization and finish message it adds up to six. Instead it was decided to send data structures with JSON.

Since the system is supposed to handle installation of applications from a device to the head unit, an application package has to be published as a MQTT message. To protect the binary data from corruption when being converted between strings and JSON objects on the different platforms it was decided to use Base64 encoding. This increases the size of the message of course, since each byte is converted to a Base64 representation but it also assures that the system will not destroy data in regular string-to-JSON conversions.

6.1.3 SERVER

The broker is the central part of the communication system. All published messages are sent to the broker so that it can forward them. The broker also handles the topic subscriptions. There are a few different MQTT brokers that could be used;

- IBM Really Small Message Broker
- Mosquitto
- Node.js

Since the IBM broker is not open source it was out of the question for this project. Mosquitto is a well known open source broker written in Python. Cybercom had experience from using Mosquitto in other projects so for research purposes it was decided that this project should try to implement the broker using Node.js. There is also an MQTT plugin for Node.js which is called mqttjs that could be used in the implementation. If it doesn't work out, it will be possible to use the Mosquitto broker as it is.

SERVER OVERVIEW

The server was written in Node.js and is the central application for the communication. It handles all incoming messages and forwards them to the correct receivers. It contains both MQTT broker and WebSocket server where the broker is primarily responsible for the communication and the WebSocket server contains a connection bridge between WebSocket and MQTT. In the architecture of the project the server is installed in the computer system of the car with Wi-Fi connection.

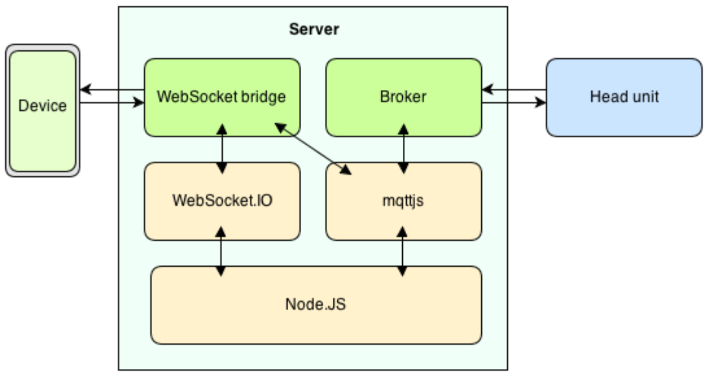

Figure 6: Preliminary server design

BROKER

Mqttjs is a tiny Node.js plugin which allows the user to write custom MQTT brokers and clients. It is well documented and has a few example implementations which served as a solid foundation to implement the broker on.

WEBSOCKET BRIDGE

An additional way to communicate with the broker was implemented through bridging WebSocket with a MQTT client. This made it possible for an application to communicate with the broker through WebSockets. The WebSocket server created a unique MQTT client for each WebSocket connection which then translates and forwards the messages back and forth. The MQTT client will be used for publishing and subscribing to topics in the broker and the WebSocket for the communication with the device.

6.1.4 HEAD UNIT

The head unit will be simulated with an Android application that will be used to host and show web applications to the end user.

Communication with the broker is done through a MqttClient which has been implemented with the aid of the Eclipse Paho library. The Paho library contains a series of implementations of open standardized protocols that can be used for machine to machine (m2m) messaging. Using this library will greatly simplify the development of the application since it is tested and confirmed stable by the Eclipse foundations m2m group [8].

The head unit must also be able to receive a message which contains an installation package in the form of a Base64 encoded zip file. When a message with an install action is received the head unit should read the data from the message and convert it to a ZipInputStream. Once this stream is set up it is possible to decompress the data that has been received.

The file is decompressed into a folder on the external storage of the Android tablet which is running the head unit application. The web server which is running on the tablet uses this folder as the web root folder.

Once the web application is installed in the head unit it can be displayed in a WebView. Unfortunately it is not enough to just load local file system applications into the WebView due to security constraints.

Although a simple web application will be displayed just fine some of the more popular JavaScript frameworks and plugins use Asynchronous JavaScript and Xml (AJAX) and XMLHttpRequests to load local files. This is not allowed in the WebView due to cross-domain security restrictions and the same origin policy [9]. Instead, a third party Android application, kWS, will be used to run a local web server on the Android tablet which will be used to serve the web applications [10].

The head unit will handle system messages and it must also be able to forward any non-system messages to the web application that is running. Therefore it was critical for the functionality of the web application that the head unit implements the JavaScriptInterface. This interface is the only contact with the system outside of the head unit that the web application has.

COMMUNICATION FLOW BETWEEN WEB APPLICATION AND HEAD UNIT

The JavaScriptInterface allows the developer of the head unit to open an API to the web application developer. The methods that are implemented in this project are;

- onMessage(String topic, String payload)

  Called when the head unit receives an MQTT message and forwards it to the web app.

- subscribe(String topic)

  Can be called by the web app to subscribe to any topic.

- publish(String topic, String payload)

  Can be called by the web app to publish a message to any topic.

- unsubscribe(String topic)

  Can be called by the web app to unsubscribe from any topic.

- showToast(String message)

  Shows a message to the user with the standard graphical components of the head unit.

Using these methods the developer of a web application can publish and receive messages from the rest of the system. The developer has the option to set up a custom private channel, which must be uniquely named, for private communication between the device- and web-application.

MODEL

The application relies heavily on the Callbacks pattern which is common in Android and follows the best practice guidelines in the Android documentation [11]. The application starts with the system launching MainActivity. This activity will initiate the WebView of the application to enable JavaScript and add a custom chrome that allows the application to hijack any logging to the console from the web application. This is mainly for debugging purposes.

Once the WebView is set up the ApplicationController may be launched. The controller starts a new MqttWorker which is running in a separate thread for networking I/O. Anytime the worker receives a message the onMessage() method in CustomMqttCallback is called which then triggers a Callback to the controller passing along the message details.

The ApplicationController also controls the Decompresser which decompresses zip data and when the decompressing is finished another callback is triggered to the controller. The ApplicationController may also show dialog messages to the user, these dialogs are build in the DialogFactory which also triggers a Callback to the controller when the user has selected an option in a dialog.

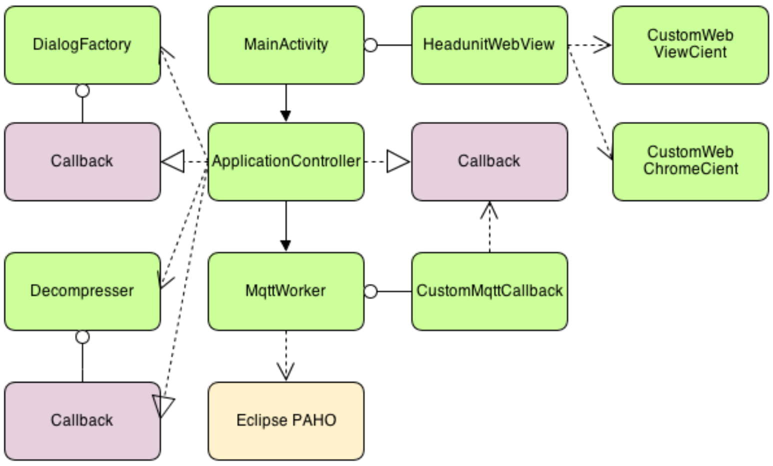

Figure 7: UML diagram over the head unit application.

6.1.5 COMMUNICATION BETWEEN PRIMARY DEVICE AND HEAD UNIT

Figure 8 shows the states of the primary device application. A developer that is creating applications for the device can change the state by publishing different messages and handle the responses. All the communication from the device to the head unit is on the '/system' topic. The head unit will respond on the private topic set by the device application.

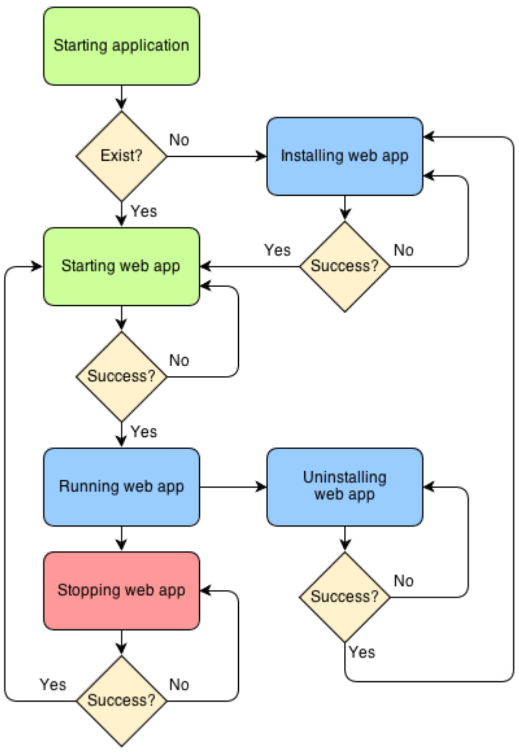

Figure 8: Flow and state diagram over the primary device and head unit communication.

Here are examples of action and response messages and reviews of all messages required.

ACTION MESSAGE

```
{
    action : 'the action',
    data : 'unique name / binary representaion of zip file'
}
```

RESPONSE MESSAGE

```
{
    action: 'the action',
    data : 'success / error',
    type, 'response'
}
```

EXIST

The exist message must be published from the device on the '/system' topic with unique name in the data field, this name will then be used as the applications private topic. The head unit will receive the message and respond with a success message if the application is already installed. If there is not an application installed the head unit will respond with an error message. Depending on the response message from the head unit, the device has three alternatives;

- push and install the web application if the response includes an error message
- start the web application if the response contains a success message
- uninstall the web application if the response contains a success message

INSTALL

The install message must include a compressed version of a web application in the data field with the action install and be published to the '/system' topic. When the head unit receives this message it will first respond to the device´s personal topic with the action 'pending' and then start to decompress the application and store it in its local memory.

When the application process is successful the head unit will publish another response message to the device´s private topic that contains success in the data field. The device now has the two options of starting or uninstalling the web application. If an error occurs during the installation process the response message will include 'error' in the data field and the device must send another install message before a new attempt to install is possible for the head unit.

START/STOP

The start message is published to the '/system' topic and includes start as the action and the unique name in the data field which decides which application the head unit will start. If the application is successfully started a response message with 'success' in the data field will be published to the device private topic. This message allows the device to start communicating with the web application by publishing to their mutual topic.

The only message it can publish to the head unit at this point is a stop message which has the same structure as the start message except the data field in which 'stop' must be included. The head unit will respond with success in the data field if the application is stopped and error if something went wrong. If the stop message response is successful the device has the options of starting or uninstalling the web application.

UNINSTALL

An uninstall message will include 'uninstall' as the action and the unique name in the data field, it must also be published to the '/system' topic. When the head unit receives this message a response will be published to the device´s private topic with 'pending' in the data field and 'uninstall' as the action. The head unit will then delete the application and when it is done a new response will be published with either 'success' or 'error' in the data field pending on the success of the deletion. If the device receives an error response a new 'uninstall' message must be published before the head unit can attempt to delete the application again.

6.2 DESIGN OF DEMONSTRATION SYSTEM

The project requires five different applications to be created;

- server that connects applications with a broker and WebSocket server
- sensor gateway to simulate sensors in the vehicle
- primary and secondary device application that can install a web application in head unit and communicate with each other
- web application to be displayed in the head unit

These applications run on different platforms but with the use of Java it has been possible to re-use some common components such as the implementation of the MQTT client in the class MqttWorker which is critical for the MQTT communication.

6.2.1 INTERACTION BETWEEN DEVICES AND HEAD UNIT

In the use case described in the introduction, a scenario with two devices that share a playlist is described. The primary device publishes a playlist to the broker on an application specific topic and the secondary device receives the playlist when the broker forwards it. Both devices and the head unit are now able to publish actions to their common topic to control the playlist. This interaction is described in figure 9.

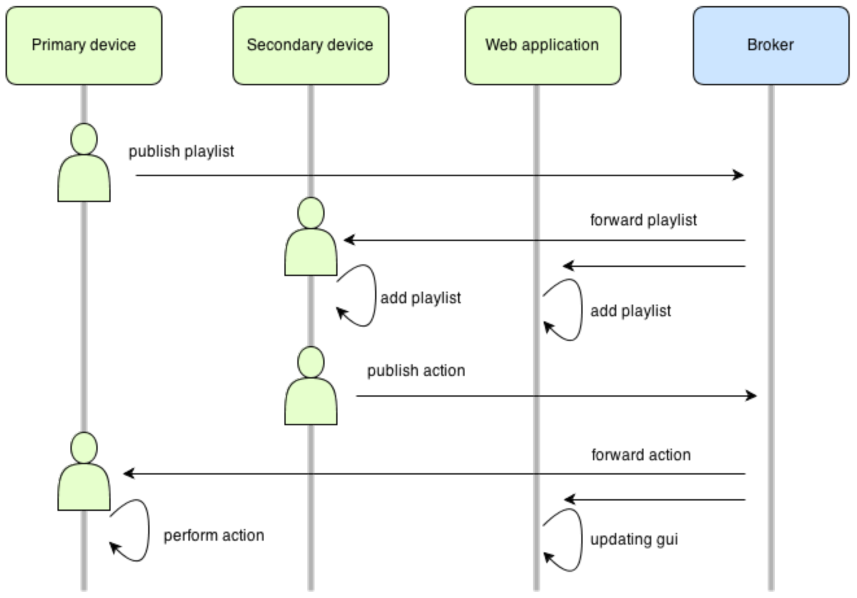

Figure 9: The primary device publishes the playlist and the secondary device receives it and can then publish actions to the primary device and web application.

ADD MESSAGE EXAMPLE

```
{
    action : 'add',
    artist : 'Any artist',
    track : 'A track of artist',
    uri : 'spotify uri',
    tracklength : 231
}
```

6.2.2 CALLBACKS DESIGN PATTERN

The system relies heavily on asynchronous work and an event driven architecture. This requires a design pattern in the applications which can handle these types of events and work done by other threads. The Callbacks design pattern is commonly used in the Android platform for example and it allows classes to delegate work to other classes which are doing asynchronous work. A class that does that kind of work has an interface with callback methods that the caller can implement. This allows the calling class to pass itself in as a callback when the worker class is being instantiated.

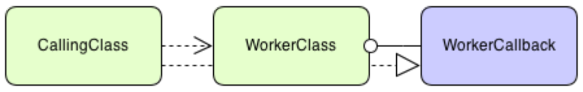

Figure 10: The Callbacks pattern is common in event driven and asynchronous systems such as Android.

6.2.3 PRIMARY DEVICE

The primary device is brought into the car environment and connects with the broker to push its web application to the head unit. It is also responsible for hosting a shared playlist and play the tracks using a Spotify library. It is developed in Android with a couple of external libraries for Spotify and MQTT.

The communication with the broker uses the library Eclipse Paho which contains an MQTT client. The client subscribes and publishes messages to specific topics with actions depending on which phase the application is in.

The first phase will include the initial communication with the head unit where the actions ‘exist’, ‘install’, ‘uninstall’, ‘start’ and ‘stop’ will be implemented. The ‘install’ method will read a zip file located in the assets folder of the project and encode it with Base64 before publishing it to the head unit.

The second phase is when the device communicates with the web application that is installed and started. It publishes and receives actions to the private topic. Another feature of this application is the playlist which can be updated through the action ‘add’ with details about a specific track. This was implemented so that another device could share the same playlist in the car and add tracks of its choosing. If another device is connected it will also have the possibility to perform actions by publishing on the private topic.

One requirement for the user of this application is to have a Spotify premium account to be able to listen to tracks.

The applications uses libspotify which is a C/C++ library. Libspotify allows a developer to take advantage of the functionality that Spotify offers and develop their own music application [12]. The main features this application will use is ‘play’, ‘pause’, ‘prev’ and ‘next’. To use a native library like libspotify, it is required to use the Android Native Development Kit (NDK).

MODEL

The application starts with the MainActivity which initiates the ApplicationController and the NfcReader. The NFC reader is used for reading an URL stored on a NFC tag and uses its callback to notify the activity. The MainActivity is responsible for the view and its listeners. It implements the Callbacks interface so the ApplicationController can notify it when an update of the view is necessary.

The ApplicationController initiates both the MqttWorker which handles the MQTT communication and the SpotifyController which is responsible for the playlist and all its functionality. Both the MqttWorker and SpotifyController use their corresponding callbacks interface for notifying the application controller. This allows the ApplicationController to perform different actions pending on the callback method called.

The SpotifyController uses the libspotifyWrapper to access the methods in libspotify, the ones used in this application will be togglePlay(), playNext() and login(). These methods use callbacks to return to the Java code in libspotifyWrapper and are important to implement to prevent application crashes.

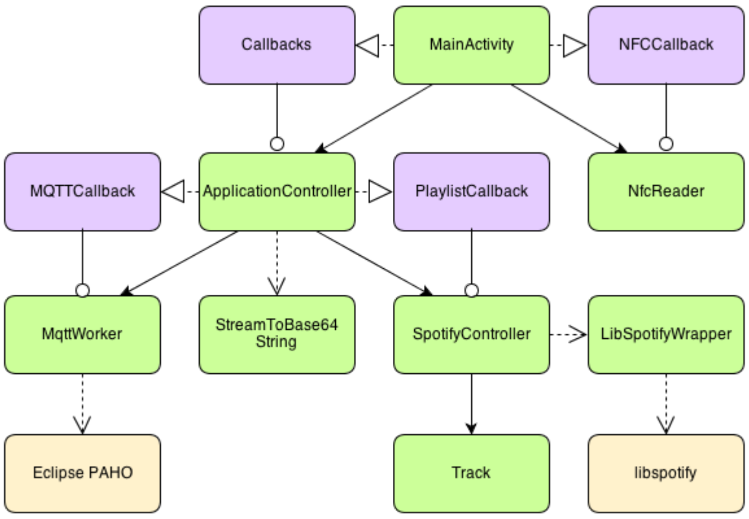

Figure 11: UML diagram over the primary device structure.

6.2.4 WEB APPLICATION

The web application shows the playlist that originates from the primary device. It updates its user interface which contains the playlist, play, pause, next and previous buttons. It is also possible to use the web application as a controller of the primary device application.

To receive actions the application uses the onMessage() function that is called by the head unit when a non-system message is received. A decision is then made inside the web application on what action to take and if any graphical component needs to update.

There are also click listeners attached to the music controls. A listener triggers an event which publishes a message on the private channel of the application containing the requested action.

MODEL

The application contains a playlist with tracks. This list needs to update when a track has finished playing or a “next” action has been received. To perform such an update in a user friendly way it was decided to use animations to show the user what is happening. jQuery is the most used JavaScript library in web applications today and it is critical to know that the head unit can handle applications written with this library. Therefore it has been chosen to implement the animations [13].

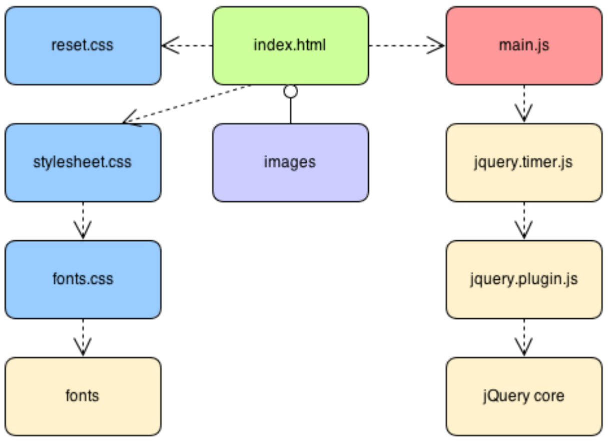

Figure 12: File structure of the web application running in the head unit.

6.2.5 SECONDARY DEVICE

The second user device in the communication system is meant to demonstrate the power of the publish/subscribe model and the usefulness of the shared display in the head unit. It takes part in the MQTT messaging through subscribing to the application topic(s) defined by the primary device. The application is a simple Android application which lists the current playlist, has controls to play, pause and skip tracks, it also has the capability of searching the Spotify metadata API for new tracks and add them to the shared playlist.

To receive and publish messages the Eclipse Paho library is used to create an MqttWorker which can subscribe to topics and receive messages with actions from the broker. The actions received will trigger events that update the corresponding parts of the user interface.

To be able to perform the Spotify search it is needed to make asynchronous HTTP requests to the Spotify servers. The Spotify Metadata API (SMAPI) contains Java bindings and an Android wrapper can be written around this library to simplify and speed up the development [14]. SMAPI makes use of the Jackson JSON Processor for the JSON to Java object mappings which is well known, stable and fast [15].

The search results are displayed in a list in which the user can click items to add them to the shared playlist. When a search result is clicked it triggers an event which publishes an ‘add’ message to the broker. Since the MQTT client subscribes to the same topic that it publishes the message to, the message will be received by the client like any other message. This means that the ‘add’ feature developed to populate the playlist with tracks in an earlier stage can be reused without any modifications.

MODEL

This is one of the more complex applications in the implementation of the demonstration tools. It starts in the MainActivity where the initial connection to the broker is made through the ApplicationController and the MqttWorker. If the connection fails, a reconnect dialog is shown, otherwise, the first page is presented to the user.

The application has three pages that are hosted in an Android ViewPager which allows the user to scroll between the pages horisontally. These pages extends the Fragment class so that it is possible to use the FragmentPagerAdapter to feed the ViewPager.

The first two pages are very similar; they extend the ListFragment class that Android provides. The first page shows search results from the SMAPI. When a search is being made, a loading dialog is shown to the user. The second page shows the current playlist. These two pages share the same class but will need separate instances since they have different content. They also share the same adapter class to feed their lists with data.

The third and last page contain the player controls that should trigger 'play', 'pause', 'next' and 'prev' actions. When such an action is triggered the ApplicationController should use the MqttWorker to publish a message to all the subscribers in the network.

The application makes heavy use of the Callbacks pattern.

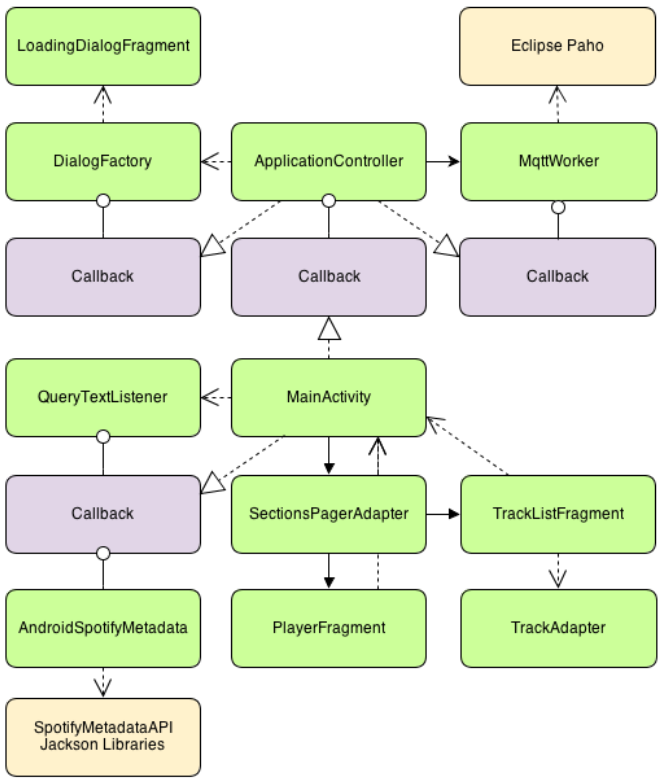

Figure 13: UML diagram of the secondary device application.

6.2.6 SENSOR GATEWAY

The gateway is simulated using a small Java application that publishes MQTT messages. The gateway in a real scenario can be seen as some type of computer which translates the CAN messages from the vehicle's electronic system into JSON data which is published to the broker. To publish these messages the app contains three buttons which each trigger an event to publish a message with the corresponding action.

MODEL

Since the Android JSON libraries are not available in Java Swing, these libraries were reused from the Android Open Source Project (AOSP). This greatly simplified the reuse of the MqttWorker that is used in other applications and saved a lot of time. The sensor gateway also uses the Eclipse PAHO library for its MQTT communication.

The Main class initiates a JFrame and add three buttons to it. These buttons have action listeners and when clicked the controller should be told. The controller then publishes a message through the MqttWorker to the broker.

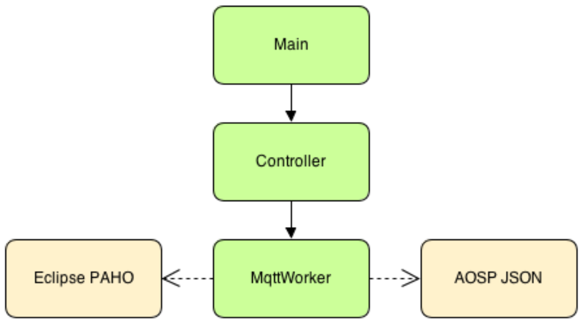

Figure 14: UML diagram of the sensor gateway simulation.

7 IMPLEMENTATION

7.1 FIRST ITERATION

The first iteration was mainly focusing on the communication of the system and therefore only the broker and head unit was implemented. It was trivial to create these applications first to see if the structure of the communication would flow without any issues. To test the communication functionality of the head unit application a simple tool from IBM was used. The tool IA92 can subscribe and publish to topics and show the contents of the messages [16].

7.1.1 BROKER AND WEBSOCKET SERVER

The server was programmed using Node.js and contains both an MQTT broker and a WebSocket server. The broker is based on an example implementation from the module ‘mqttjs’ written by Adam Rudd with a few improvements and modifications [17].

The implementation of the WebSocket bridge uses the node-modules WebSocket.io and mqttjs. A WebSocket server is created with the main responsibility to create a unique MQTT client whenever a WebSocket client connects to the server. This MQTT client is used for the connection with the broker and will forward all MQTT messages it receives to its WebSocket client. It also handles the subscribing and publishing to topics.

ISSUES

The testing of the server worked well until the discovery of an issue where the server stopped publishing messages at seemingly random intervals.

SOLUTION

The time limitations did not allow for further research to solve the issues with the Node.js server and a decision to use the broker Mosquitto was made. This broker is well known and worked as intended during the research phase which made the project to continue without any more interruptions caused by the server implementations. The WebSocket bridge did still work as intended but the server and the bridge had to be executed in separate processes since the bridge was no longer part of the server.

7.1.2 HEAD UNIT

WEB VIEW

To get started with the head unit a basic Android project was created. To make sure that the web applications would be displayed properly the WebView was the first component to be implemented. To be able to listen for events that the web view triggers when it finishes loading a page, settings were added through a custom WebChromeClient and WebViewClient. This also allowed the head unit application to hijack console.log() messages from the web application that was running which meant that it was possible to log them to the Android logcat. This was crucial for debugging purposes.

When the web application was running smoothly the JavaScriptInterface was implemented. The implementation of the interfaces went smoothly for the most part but there were some type issues that caused problems when the head unit was forwarding a message through the onMessage() function.

The onMessage() function takes a parameter containing the message to the web application. In the Android application this was treated as a string representation of a JSON object. Thus it was expected that the JavaScript implementation of onMessage() would receive a string which could then be parsed into JSON.

JavaScript interpreted this string as a true JSON object however and when JSON.parse() was called on the assumed string a non-descriptive error was thrown. It took some time of debugging to find the issue but when solved, the message forwarding was working well.

When the interface was working and it was possible to send messages between the head and the web application it was time to implement the MQTT client. The client needed to be as generic as possible since it would be used in the other Java applications as well, preferably with as little modification as possible.

COMMUNICATION

Android has strict rules about networking from the UI thread which caused some issues. A typical problem were buttons that would trigger a publish action needed to perform the publishing in the networking thread while still listening for clicks on the UI thread. Another issue was when a message came in from the broker and the UI needed to update, that update had to be done by the UI thread. These strict rules are in place for the performance of the Android operating system so they have to be adhered. It was solved using callbacks between threads and UI components.

Once the communication was working and the web view was loading html pages properly it was time to implement another critical feature for the project. The installation process where a web application is pushed from a device into the head unit and then decompressed into the web server's root folder.

Since the process had been thought through in advance the actual implementation did not cause too many issues. There were some problems with finding the correct way to decode the Base64 stream that contained the zip-file but the actual decompressing was easy with the help of standard Java libraries.

7.2 SECOND ITERATION

During the second iteration, the applications needed to demonstrate the use case were developed. These applications were created to test the project's communication system developed in the first iteration.

7.2.1 PRIMARY DEVICE

The primary device runs Android. The first thing that needed to be implemented was to handle the communication with the broker and the head unit. This was done with help of the Eclipse Paho library and the MqttWorker class created in the head unit. The communication pattern between the primary device and the head unit was implemented by simply publishing the corresponding action in the messages to the '/system' topic.

When the communication with the head unit was finished the feature of sending an installation package was implemented. A simple web application was compressed and stored in the projects assets folder for easy access. The ApplicationController uses the class StreamToBase64, which reads the zip file and encode it to a Base64 string, before adding it to the data field of the MQTT message.

At this point the UI of the application was not completed but all callbacks that would be used when a response from the head unit arrives was implemented. Updating the UI to match the state of the application on the responses forces the user of the application to follow the flow and perform correct actions.

The second feature that needed to be implemented was the communication with the web application. This was easy since the MqttWorker already worked as intended and the only addition was to publish messages with different actions and on another topic. It was decided that the actions needed were 'play', 'pause', 'next' and 'add'. The application should also listen to these actions to perform the corresponding operation.

When all of the MQTT communication worked it was time to create the features that allows the application to stream music with the use of libspotify. Libspotify is written in C code and therefore needed to be bridged to Java. With help of the project psyonspotify's C code and wiki, the application could connect with libspotify and use the features needed to fulfill the requirements [18].

A static url was used to connect with the broker and to make it more general NFC was implemented. This allows the user to connect to the car by simply touching their device with the NFC tag.

7.2.2 SECONDARY DEVICE

The implementation of the secondary device started with implementing the Spotify Metadata API search and presenting the results. There are several open source libraries available in Java that can be used to fetch data from the API so there was no need to write a custom implementation for the core functionality. Due to the Android operating system's strict rules about network communication it was necessary to write an Android wrapper that would perform the search in an asynchronous way around the library however.

Since this implementation is for demonstration purposes, the search for albums and artists was not implemented. Instead it was sufficient to search for tracks.

Once the asynchronous search returned a result, an event would be triggered which added the search results to an Android ListView. To be able to add tracks from the list into the actual playlist, click listeners were added to each row of the list which would publish an 'add' message to the private channel of the primary application. The message would then be distributed to all the subscribers who would add the track in the message to their playlists.

The user of the application should also be able to see the playlist in the same application that performs the search so another page was added to the application using a ViewPager and Fragments. The use of Fragments enabled reuse of the ListView that shows the search results and using the ViewPager made it easy to extend the application with more pages.

The last page that was added was the playback control page. It simply contains buttons for the 'play', 'pause', 'next' and 'prev' actions. When a button is clicked the controller of the application will ask the MqttWorker to publish a message to the primary application's private topic with the specific action.

The user can connect to the broker either by scanning an NFC tag that holds the URL to the broker or by entering the URL manually.

7.2.3 WEB APPLICATION

The web application which the primary device would push into the head unit did not cause any big problems. The requirements were well defined and since the head unit had been built already it was not hard to simply implement the functions defined in the JavaScriptInterface to allow the web application to communicate.

There were two main features which the application needed to fulfill; show/update a playlist and control playback via some clickable buttons. The playlist was easy to implement as a simple unordered list with some styling. The list updates when the onMessage() function is called with an 'add' or 'next' action, with the help of jQuery's excellent DOM manipulation it required little effort.

The clickable buttons were as easy to implement using the jQuery.click() method and once a click is registered the JavaScriptInterface.publish() method is called to publish the appropriate message.

Using jQuery as the core JavaScript library turned out to be a great choice. jQuery has an easy to understand syntax and the chaining of method calls allow for short, flexible and readable code. The jQuery plugin "timer" was used to control the progress bar animation. There was not enough time in the project to implement a proper scrubber nor synchronization of the progress over time.

7.2.4 SENSOR GATEWAY

To simulate the sensors a tiny little Java program was created. The program simply creates a JFrame and adds three JButtons to it. Each button is supplied with an ActionListener which triggers a call to the ApplicationController which publishes a 'play', 'pause' or 'next' action on the private channel of the primary application.

To publish messages, the controller has an instance of the MqttWorker that was first created in head unit. The class has been cleaned up a bit since the sensor gateway does not listen to any communication and it only publishes simple messages. The MqttWorker was originally written for Android so the Android specific things have been removed from the earlier implementations of the class. To be able to reuse as much as possible of the class it was decided to use Android's org.json library to parse JSON.

8 RESULTS

The project has been carried out in three phases, research, analysis and implementation. The research phase gave good clues on how an embedded system like the one in a motor vehicle can be connected to the devices around it.

8.1 COMMUNICATION ARCHITECTURE

The publish subscribe model that is offered by MQTT was a great solution for a communication system where clients and nodes are constantly changing. Another great feature that MQTT offers is that messages are not only being sent to a single client. The possibility of having one-to-one and one-to-many communication creates a solid foundation and a very flexible communication platform for other applications to use. This is much more flexible than the common request/response model that HTTP uses.

8.1.1 COMMUNICATION

Depending on what type of device is being used the communication between device, head unit and web application can differ. There are libraries available online that allows a developer to create MQTT clients with minimal effort for many different languages where Java, Objective-C and JavaScript are only a few.

There seem to be some issues with the technology of hybrid applications however. The tools and platforms that are commonly used like Phonegap and IBM Worklight can be tricky to use. This project was not successful in adapting these tools together with HTML5 technology to develop device applications that could communicate with the head unit over MQTT or WebSockets. It certainly might be possible but there was not enough time to thoroughly examine these possibilities, neither was it part of the original goals of the project.

8.2 INSTALLATION

The installation process was critical for the project to succeed, would it be possible to push web applications from one device to another? In this report it has been shown that it is possible, it is not even difficult. MQTT offers the possibility of sending large chunks of data in any form as long as there is an input- and output-stream, applications can be sent between devices. The HTML5 applications that were sent, installed perfectly and ran without issues in the head unit.

One issue that arises is the display of these applications. The web application technology often assumes and relies on that the content is being served by a web server. The complete browser model is based on this assumption and the security restrictions that are in place requires that an implementation follow the browser's rules. It is most likely possible to implement a custom web view which does not have these restrictions in place although web servers are available as free and open software so they might as well be used.

9 CONCLUSION

9.1 COMMUNICATION AND IMPLEMENTATION

During the development of the project we have explored a couple of different ways of creating an implementation of the MQTT communication system with web technologies. It seems that web technology is suitable for some parts of such an implementation but the technology has not matured to a point where it is possible to create the applications with pure HTML5 without resorting to native applications. Even with the use of frameworks such as Phonegap it was not trivial to connect to a WebSocket server nor to an MQTT broker from the device. Unfortunately we had to abandon these implementation ideas due to time limitations.

Using native technologies for developing the device applications has solved these issues and it was very easy to implement a working web application which was being hosted by the head unit. Since we used an Android application to simulate the head unit we got the web view for free but it should not take too much effort to use a web view from Qt or some other framework.

It seems that the machine to machine (m2m) technology is mature enough and that it does not have to be difficult. The question of how to implement such a system in a secure way remains and should be investigated further.

There is also a possibility to look at the communication system from the other side. In our implementation, we have pushed web applications to the vehicle. It would also be possible to push applications from the vehicle into a device. Such a system would allow OEM's to deliver manuals, service information, driving data and debugging information to the device. This would certainly be useful for both fleet and private owners.

9.2 SECURITY IMPLICATIONS AND SOCIAL IMPACT

Security issues can have a big social impact on the user if the wrong kind of information gets into the wrong hands. There are many different types of security implications in a system like the one that has been developed. There are the obvious implications of the driver being distracted but there are also other problems related to the implementation itself. This is by no means a security analysis of the system that has been implemented. Some key problem areas should be pointed out though.

9.2.1 AUTHENTICATION AND AUTHORISATION

In this implementation there is no authorisation taking place at any time. A device would certainly have to prove its identity to the vehicle before it could be allowed to take part in the communication. There is also an issue of which device is allowed to install applications in the head unit and which device can only listen. There would most likely have to be some sort of privilege hierarchy in a real implementation.

9.2.2 ENCRYPTION

The data that is being sent in this example implementation of MQTT is never encrypted. MQTT offers support for SSL/TLS which could be used to ensure secrecy of the messages that are being published.

9.2.3 SANDBOX BREAKOUT

When running a web server in a car, all the problems of the web enter the car. It could be possible for installed apps to contain malicious code that could break out of the sandbox that the head unit should offer. If persistent storage is added using databases, all the database related issues such as SQL injection and XSS would come into play.

9.2.4 INFORMATION LEAKAGE

Any MQTT client can subscribe to any open channel. In this implementation only open channels have been used. However, a broker might offer the possibility to password protect topics and publications. Mosquitto which was used in this project is one example of such a broker.

If the applications running in the system are not properly implemented there would most likely be applications that have malicious intents. Such applications could spy on the user, reporting location data and other sensitive information to an unknown party.

9.3 ECONOMICAL AND ENVIRONMENTAL IMPACT

There are not many directly connected sustainability issues with developing a software product like the one that has been developed in this project. However, there are some points that should be taken into consideration when developing any computer system.

Developing software does not necessarily has direct impact on social sustainability but software development requires tools. The tools used could very well have a negative impact on the workers that produce the components used in a computer. The electronic industry is well known to push workers hard and pay salaries below a living-wage [19].

When analysing the use of electronical components it is important to take into account both the computer that is used for developing the software and also the computer that is used to run the software by the end user. It should also be noted that the use of electronical components in computer systems require resources that are not endless. Any use of a limited resource takes its toll on the environment and produces waste.

In a car environment this means that the manufacturers will have to take responsibility for the electronic parts they use in their vehicles and make sure they are produced in a sustainable way.

It is possible to create application that might benifit the environment as well however. A fleet of vehicles can report real time data from the vehicle to some central server while at the same time letting the drivers know the fuel comsumption. This could easily be used in competitions between drivers where the most eco-friendly driver will be rewarded. A system like the one developed would also open up a new market for economic growth since application developers and service providers would be able to sell more software and services.

9.4 FINAL WORDS

We think that it is inevitable that the car will be an open platform at some point in the future. Users today are demanding personalisation and customisation in their products. With the smartphone explosion new possibilities and demands have been created where people expect to be able to use their content anywhere, anyhow.

Will the OEM's push for such a change though? Do they really wish to open up their platforms to third party developers without control?

In today's infotainment systems we see applications like Spotify, Pandora, Mail and more but they are all controlled by the automotive manufacturer. In our opinion it would be a great selling point to offer an open platform in which the owner/user of a vehicle are free to install whatever applications they wish. Such an open platform would most likely shorten the development time of new applications which could be hosted in the vehicle.

BIBLIOGRAPHY

[1] Eurotech International Business Machines Corporation (IBM). Mqtt v3.1 protocol specification. https://public.dhe.ibm.com/software/dw/webservices/ws-mqtt/mqtt-v3r1.html, May 2013. (Cited on page 5.)

[2] Valerie Lampkin, Weng Tat Leong, Leonardo Olivera, Sweta Rawat, Nagesh Subrahmanyam, and Rong Xiang. *Building Smarter Planet Solutions with MQTT and IBM WebSphere MQ Telemetry*. IBM, Armonk, NY, USA, 1st edition, 2012. (Cited on page 6.)

[3] Mosquitto.org. An open source mqtt v3.1 broker. http://www.mosquitto.org, May 2013. (Cited on page 6.)

[4] Pedro Teixeira. *Professional Node.js: Building Javascript Based Scalable Software*. Wrox, 1st edition, 2012. (Cited on page 6.)

[5] Robin Berjon, Travis Leithead, Erika Doyle Navara, Edward O Connor, and Silvia Pfeiffer. Html 5.1 protocol specification. http://www.w3.org/TR/html51/, May 2013. (Cited on page 6.)

[6] A. Melnikov and I. Fette. The websocket protocol rfc 6455. http://tools.ietf.org/html/rfc6455, 2011. (Cited on page 6.)

[7] Janessa Rivera and Rob van der Meulen. Gartner says asia/pacific led worldwide mobile phone sales to growth in first quarter of 2013. http://www.gartner.com/newsroom/id/2482816, May 2013. (Cited on page 7.)

[8] M2M eclipse group. Paho - open source messaging for m2m. http://www.eclipse.org/paho/, May 2013. (Cited on page 13.)

[9] Robin Berjon, Travis Leithead, Erika Doyle Navara, Edward O Connor, and Silvia Pfeiffer. Html 5.1 protocol specification. http://www.w3.org/html/wg/drafts/html/master/browsers.html#security-nav, May 2013. (Cited on page 14.)

[10] Kamran Zafar. kws - android web server. https://play.google.com/store/apps/details?id=org.xeustechnologies.android.kws&hl=en, May 2013. (Cited on page 14.)

[11] Android Developers. Dialogs. http://developer.android.com/guide/topics/ui/dialogs.html, May 2013. (Cited on page 14.)

[12] Spotify Ltd. Libspotify. https://developer.spotify.com/technologies/libspotify/, May 2013. (Cited on page 19.)

[13] w3Techs.com. Usage statistics and market share of javascript libraries for websites. http://w3techs.com/technologies/overview/javascript_library/all, May 2013. (Cited on page 20.)

[14] Hector Izquierdo. spotify-metadata-api. https://github.com/hekoru/spotify-metadata-api, May 2013. (Cited on page 21.)

[15] codehaus.org. Jackson json processor. http://jackson.codehaus.org/, May 2013. (Cited on page 21.)

[16] Eurotech International Business Machines Corporation (IBM). Wbi brokers - java implementation of websphere mq telemetry transport. http://www-01.ibm.com/support/docview.wss?uid=swg24006006, May 2013. (Cited on page 24.)

[17] Adam Rudd. Mqtt.js. https://github.com/adamvr/MQTT.js, May 2013. (Cited on page 24.)

[18] Spotify Ltd. psyonspotify. https://github.com/spotify/psyonspotify, May 2013. (Cited on page 26.)

[19] Debby CHAN Sze Wan and CHENG Yi Yi. *Workers as Machines: Military Management in Foxconn*. SACOM, Hong Kong, 1st edition, 2010. (Cited on page 30.)