Marek Krajewski

Cross-platform development of the Smart Client application with Qt framework and QtQuick

Information Technology
2016

ACKNOWLEDGMENTS

I would like to thank Dr. Smail Menani for giving me opportunity to participate in this project, guidance during the thesis, his support and extreme patience.

I would also like to thank Rafał Chomentowski for always being willing to share his expertise in the Qt Framework.

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Degree Programme of Information Technology

ABSTRACT

Author	Marek Krajewski
Title	Cross-platform development of the Smart Client application with Qt framework and QtQuick
Year	2016
Language	English
Pages	62 + 1 Appendix
Name of Supervisor	Smail Menani

In this thesis the Qt Framework is evaluated as the tool that can support the cross-platform development of desktop, mobile and embedded applications. Hence, a hybrid client application is developed to assess its capabilities for creating a product providing a good user experience on a wide range of the target devices. The application is required to demonstrate implementation of the Graphical User Interface, network communication with a server and access to the native development environment of the target device while utilizing tools bundled with the framework. The application is successfully developed and tested on the following devices: Windows notebook with the full size desktop monitor, Android devices with 5-inch and 10-inch touchscreen displays, Raspberry Pi with Raspbian Linux and full size desktop monitor. The QML language is used to create a responsive GUI, utilizing diverse collection of widgets provided by the QtQuick library. Qt API itself is sufficient to create a WebSocket communication with the server and allowed for leveraging the native SDK of each tested platform. A custom cross-compile toolchain is built and used in the development for the Raspberry Pi.

The result of this work proves that the Qt Framework is a feasible solution for the cross-platform development for experienced teams, offering powerful GUI creation tools and wide range of supported platforms.

Keywords Cross Platform Development, Qt, QML, Hybrid Client

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF ABBREVIATIONS

- PLC – power line communication
- HMI – human machine interface
- PC – personal computer
- GUI – graphical user interface
- UI – user interface
- HAN – home area network
- WPAN – wireless personal area network
- IDE – integrated development envirnonment
- QML – Qt modeling language
- NFC – near field communication
- AJAX – asynchronous JavaScript and XML
- SSH – secure shell
- SSL – secure sockets layer
- ADB – Android debug bridge
- API – application programming interface
- RIA – rich internet application
- GPL – general public license
- LGPL – lesser general public license

LIST OF APPENDICES

APPENDIX 1. Source code

1 INTRODUCTION

This thesis started as a part of the larger project researched and developed under supervision of Smail Menani, D.Sc. at Technobothnia, Vaasa; which is focused mainly on the concept of the Smart Grid and Smart Home. The main goal of the project is development of a prototype of the working solution able to provide a two ways communication between the typical end-consumer in the electrical network and the rest of the grid's components.

One of the required elements in the project was the client application for the end-consumer, allowing for managing the power consumption, viewing the current status and, to a certain degree, also controlling all connected home appliances.

Chapters throughout this document are organized in three main sections. The first section introduces reader to the background concepts of the Smart Grid, Smart House and the actual focus of the thesis: the Smart Client. The second section discusses findings of the research of challenges related to the cross-platform development and how Qt framework can help to overcome them. The third section goes through the design decisions made based on conducted research, documents the development of the application prototype and evaluates Qt framework as the cross-platform development tool.

1.1 Smart Grid

"In short, the digital technology that allows for two-way communication between the utility and its customers, and the sensing along the transmission lines is what makes the grid smart." /1/

Quote above shows the key point of the smart grid - the electrical grid that is self-aware and provides ability to exchange information between all actors it consists of: a power plant, substation or even the end consumer paying the electricity bill.

Some of the practical implementations of the Smart Grid are /1/

- Reduced peak demand of the electricity by a better management of the power consumption. For example, scheduling of various home appliances or loading electrical cars during the time of the day when electricity demand is the lowest.
- Prioritizing electricity to the mission critical consumers like hospitals.
- Smart Home - concept with already existing solutions utilizing the Power Line Communication (PLC) and Home Area Network (HAN) to connect and manage appliances allowing for lowering energy bills by scheduling them according the energy prices throughout the day.
- Smart Metering - solution already being adopted by many energy companies allowing for the remote, real time consumption measurements, eliminating costs of the labor related to collecting measurements manually.

At the moment, the Smart Grid has still a very futuristic sound to it; however, with the growing demand and the aging infrastructure it's already a serious topic attracting interest of organizations like Institute of Electrical and Electronics Engineers /3/ and International Energy Agency /4/. Furthermore, the Smart Grid concepts are already being deployed around the world by, for instance, Italian electricity manufacturer Enel /5/

1.2 Smart Home

The Smart Home is a concept of introducing the residential customer of electrical network to the Smart Grid by utilizing technologies like Power Line Communication (PLC) and Wireless Personal Area Network (WPAN). Such infrastructure can be also used further to implement the home automation concepts. /2/

The following diagram shows a design of the system prototype, developed in Technobothnia; which tries to combine features of both: the home automation and The Smart Home.

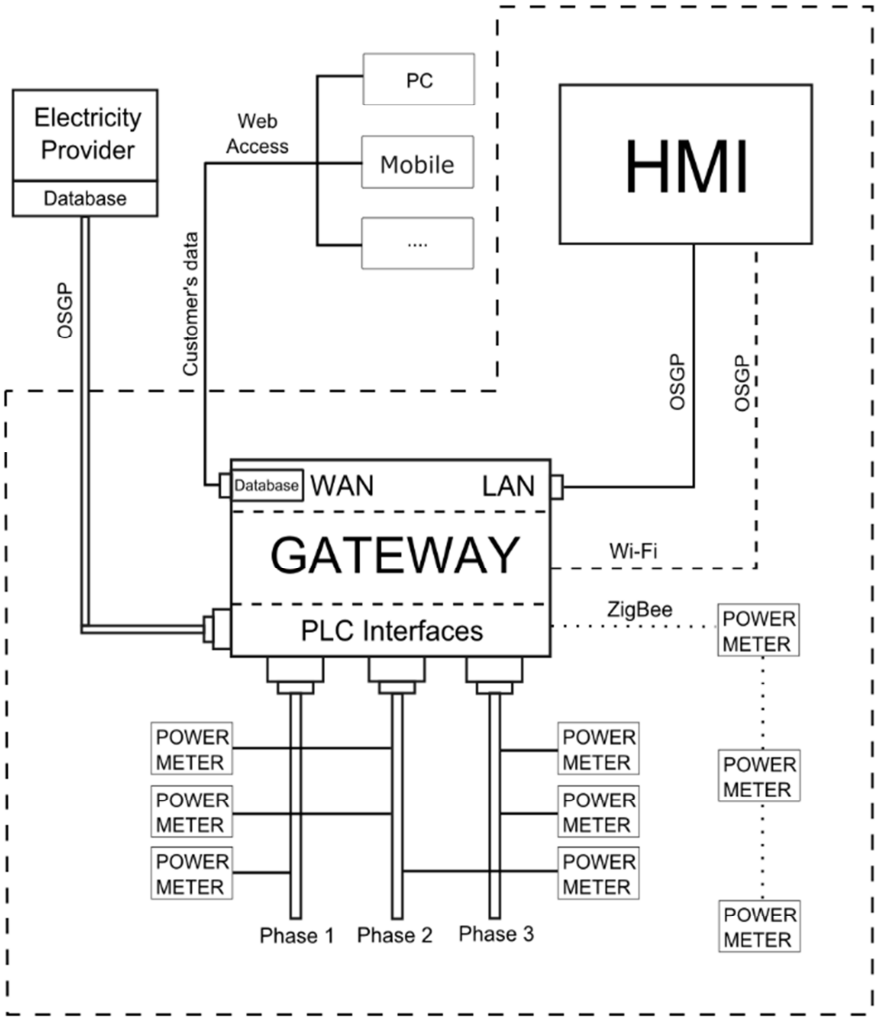

Figure 1 Overview of the project developed at Technobotnia.

Each of the home appliances is equipped in the power meter and the module capable of providing communication through PLC or WPAN solution (e.g. ZigBee) to the Gateway. The Gateway device has multiple purposes. First of all, it is connected through PLC modules to all phases of the electrical grid available in the domestic property. Together with the wireless communication module like ZigBee, the Gateway gathers the power consumption data and controls all appliances connected to the power grid.

Customer can view collected data and control the appliances through the Human Machine Interface (HMI) located in a convenient place within home area, or using

other devices connected via the Internet network: personal computer, PDA or Smartphone.

The Gateway can also exchange data with the electricity provider using the communication over the power grid or the Internet network. It allows the customer to see information about the current electricity price, billing status and notifications of the potential issues occurring in the grid. At the same time, the electricity provider can gather collected data remotely and monitor grid for malfunctions.

1.3 Smart Client

Having established the background of the thesis, it’s now possible to bring into the picture its actual subject: The Smart Client.

In the context of The Smart Home solution presented in the previous chapter, The Smart Client is an application running on HMI, Smartphone or PC capable of providing the customer an interface to the home automation, the data gathered on the Gateway and the communication with the electricity provider.

As for this moment, the exact specification of the services offered by the Gateway are not yet defined which makes the complete implementation of the client application impossible. However, the key requirements are already known and can be used to assess which technology should be used to develop the Smart Client application.

1.4 Statement of the problem

Problem: Evaluate whether Qt framework can be successfully used for development of the Smart Client application. Base your evaluation on the Smart Client requirements by:

1) Investigating the challenges related to each requirement.

2) Finding and choosing the best solution offered by Qt that can fulfill the requirement. Base the research mainly on the official Qt documentation.

3) Implementing a simple application that proves that the chosen solution satisfies the requirement.

After implementation is finished, summarize which goals could and could not be accomplished and why. Compare them with the ideal solutions found during the investigation phase.

Requirements:

<table>
<tr><td colspan="2">1. Cross-platform support</td></tr>
<tr><td>Research goals:</td><td>1.1 Evaluate the possible approaches to the cross-platform development. Find the best one, considering all other requirements of the Smart Client application.

1.2 Find if Qt framework, according to the official documentation, can be used for the application development targeted on the following platforms:

• Desktop : Windows, Linux, Mac OS X
• Mobile: Android, iOS
• Other: custom embedded device using Linux and external display</td></tr>
<tr><td>Implementation goals:</td><td>1.3 Deploy application on: Windows, Android and Raspberry PI device.

1.4 Application must contain the implementation goals specified by all other requirements (unless stated otherwise or re-</td></tr>
</table>

	quirement is not applicable for the particular platform).
2. Graphical User Interface	
Research goals:	2.1 Evaluate solutions offered by Qt framework that can be used to develop the Graphical User Interface. **2.2 Evaluation must cover the following aspects:** • Available controls. • Look and feel of the controls. How difficult it is to achieve the native look and feel of the target platform? • Different screen types and screen sizes. How difficult is it to provide GUI usable on all screen types and sizes?
Implementation goals:	2.3 Implement GUI that demonstrates usage of: • Charts. • Lists • Buttons. • Multiple views – must consist of at least two views and navigation between them. 2.4 Demonstrate that GUI can be tailored to accommodate for the following screen types: • Full size desktop monitor (17-25"), using mouse and keyboard as input. • Typical Smartphone display (5") with touchscreen and both: landscape and portrait orientation. 2.5 GUI does not have to be esthetic; however, must be usable: • Size of the controls adjusted to the screen size • Support for the common gestures when using touchscreen (e.g. swipe)

	• Responsiveness

3. Client-server communication	
Research goals:	3.1 Research what technologies and solutions can be used with Qt framework to provide the communication between the Smart Client and Gateway. Chose most appropriate one for the Smart Client and use it when developing prototype. Justify the choice.
Implementation goals:	3.2 Implement a simple Gateway emulator using the chosen technology to implement the communication between the Gateway and the Smart Client. 3.3 Data provided by the Gateway should change over time to demonstrate the continuous values update. 3.4 Implement the same communication technology in the Smart Client application. 3.5 Demonstrate that communication between the Gateway and Client works by showing: • Single request • Continuous request • Bi-directional requests (that can originate also from the Gateway)

4. Native access of the client device	
Research goals:	4.1 Research the possibilities of accessing the native development environment of the client device using following systems: Windows, Linux, Android
Implementation goals:	4.2 Demonstrate the native access on Android by implementing any functionality using the Android SDK API.

5. Development tools	
Research goals:	5.1 List and shortly describe tools offered by Qt framework that can help in: • Writing code • Debugging • Designing user interface • Cross compilation • Deploying application on the target device
Implementation goals:	5.2 Use the Qt Creator and other offered tools during the prototype development, and assess their usability by comparing it to other tools used throughout the university projects. 5.3 Build a Qt cross compiler on the Linux machine. Use it to build the Qt framework binaries and Smart Client application for the RaspberryPi2 device.

2 PROBLEM ANALYSIS

In order to assess and choose the right technology to meet defined requirements it's necessary to first discuss concepts related to the development of a client application. This chapter tries to identify challenges that will have to be addressed during the implementation phase using Qt platform, and chose the best tools and design that can help to successfully build a working client prototype.

2.1 Why Qt/QML?

Qt is a cross-platform framework allowing for the application development using the C++ language, and deployment on the most of major platforms and operating system like: Windows, Linux, Android and iOS. It comes with the Commercial and Open Source licensing options. The Open Source version is available under GNU Lesser General Public License (LGPL) /6/ which is especially important in low-budget, educational project like this one. In addition, even if runtime libraries for the target platform are not directly supported and available on Qt website, it is possible to compile a custom toolchain from the source code. Qt wiki page contains a comprehensive guide explaining the process of cross compiling Qt applications for Raspberry PI device. /7/

Qt framework isn't the only available technology that could potentially be used in the Smart Client project. We have to admit that the motive behind choosing Qt for evaluation in this thesis was partially our previous experience with it. Most of the Smart Grid project members are students of the IT degree focused on the embedded software development. Hence, it's understandable that most of them will prefer working with a framework utilizing the familiar C++ language; rather than more web-oriented technologies used by the alternative solutions.

2.2 Client archetypes

An important aspect that has to be considered before client implementation is its archetype. Microsoft's Application Architecture Guide discusses this topic in detail presenting advantages and disadvantages of each solution /8/

- Rich client application – typically, a stand-alone application that can utilize the network connection for accessing the data, and contains most the actual business logic. Requires the installation on the client, dedicated for each supported platform, which increases difficulty of providing regular updates. In return, gives access to the client's resources and allows for writing a responsive and rich GUI.
- Web application - the whole application is fetched from the server and executed in the web browser. This eliminates the problem of the application versioning and makes it accessible for all platforms equipped with a web browser. Downside is a slow, restricted user interface; dependence on the network access and the very limited access to the client resources.
- Rich Internet application (RIA) - similarly to the web application, the RIA is loaded from the server but executed in a dedicated sandbox. Middle ground between the Rich Client application and Web application. Simplifies application updating, and allows for building rich user interfaces and better access to the client resources. However, the client must contain the required version of the sandbox which forces user to be aware of the versions compatibility.

In addition, Sten Anderson, in his critique of desktop (rich client) and web applications /9/ presents yet another type of the client application: Hybrid. On the first glance, the Hybrid application resembles RIA, but provides support for the different view implementations:

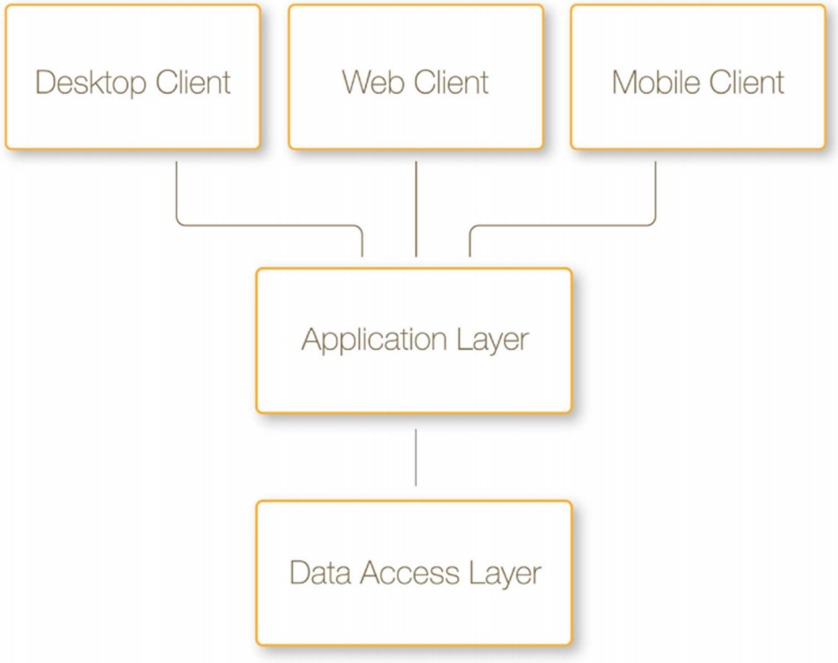

Figure 2 Three-level architecture of Hybrid Client application with multiple views implementation. /9/

The multiple view implementations give the hybrid archetype a strong advantage when application has to be supported on both: desktop and mobile clients. Considering the differences in the screen sizes between the typical desktop and mobile device, and the way the user interface is accessed (touch screen vs mouse pointer/keyboard), it's virtually impossible to provide a usable user interface with just a single view.

According to the Qt documentation, it offers the following means to implement the presented client application archetypes:

- WebApplication - Although Qt itself is able to handle displaying the content like HTML and JavaScript via QtWebEngine deployed on the client, it does not offer any tool to work directly with standard web browsers.

- Rich Internet application – The QtQuick offers possibility to display the remote QML files using QtWebEngine together with other web content. In addition, any C++ class extending QObject can be exposed, either by using QWebChannel or qmlRegisterType function (for QML only), giving the access to the client resources and additional libraries written in C++.
- Hybrid – Similar to RIA, uses the QtWebEngine. However, different views are served by the server depending on the client device type.
- Rich Client application - Similar to RIA, QtWebEngine can be used with QML files, bundled with the client application. Alternatively, the Qt Widgets module offers a set of common desktop widgets that can be used to build the GUI directly from the C++ code.

2.3 Cross-platform support

Developing an application targeted to more than one platform presents a set of challenges. Each operating system offers own environment that allows for deploying applications, accessing system resources, rendering user interface, etc. It might impose using certain programming language or compiler, depending on architecture of the processor on the host device. This chapter will outline two basic approaches to the cross-platform development and then explore how Qt framework deals with it on the supported platforms.

Applications can be developed separately for each targeted platform using its specific development environment. For example, Windows application will be written in C/C++ utilizing Win32 API and MFC libraries and compiled with Mi-

crosoft C++ Compiler. Same application, when ported to Android, will use Java language and Android SDK. Just the fact that two different languages are used will force developers to maintain two, completely separate code bases for each platform. On the other hand, utilizing the SDK of the operating system gives access to all of its resources and preserves the native look and feel of the graphical user interface. This approach is commonly referred to as the native development /10/.

To decrease the overhead related to the maintenance of multiple codebases the platform specific functionalities can be abstracted, producing code that is shared between all supported platforms while using different implementations of those functionalities. For example, the application wanting to display a graphical user interface could leverage toolkits like GTK+ offering a single interface, in wide range of popular programming languages, and separate runtime environments for Windows, Linux and Mac OS X. /11/

Qt framework works in a similar fashion providing developers a set of modules like QtGUI - for graphical user interface development or QtNetwork - for network programming. Officially supported languages are: C++, QML and JavaScript; however, additional language bindings are available with projects QtSharp /12/ or PyQt /13/ Once deployed, application uses Qt runtime environment, installed on the client, to get the actual implementation of used functionalities.

The full list of platforms supported by Qt can be found on official website, which includes: iOS, OS X, Android, Windows, WindowsRT and Linux. Depending on the operating system, it’s also possible to use the alternative solutions for handling input and display management like: EGLFS, linuxFB, XCB. This is especially useful when application is deployed on the custom embedded device with limited processing power that uses more lightweight solutions than X11 windowing system.

2.4 Native access

Occasionally, client applications may want to use the client's resources and functionalities other that just accessing the user input and rendering the user interface. As mentioned in chapter 2.2, those resources are greatly limited by the web browser for the Web Applications, mainly for the security reasons. Rich applications, quite contrary, can access the features like taking pictures with camera or reading measurements of the gyroscope by directly accessing API offered on operating system or using a third party library to do so. Rich Internet and Hybrid applications are restricted only to the features provided by the sandbox they run on.

Most of the frameworks, used to develop the Hybrid applications, offer a variety of features sufficient for most of the mobile applications including: the camera access, sensor measurements, push notification, and others. In case when application has to use features specific to the particular device, or access a third party library, some frameworks provide option to extend the basic API. For example, a popular mobile framework PhoneGaps allow for registering the custom plugins, which can be then accessed from the client code written in JavaScript/HTML. /14/

Qt framework is no different. Acting as sort of sandbox, the QtWebEngine is capable of displaying QML code having access to a vast range of functionalities offered by QtQuick library in form of the QML types. Each QML type is essentially a specially formatted C++ class, registered in the QtWebEngine. For example, to use a proximity sensor, the QML file has to simply import a QtSensors type. Custom extensions can be provided in a similar way to the QtWebEngine, giving access to the functionalities not supported in QtQuick or even possibility to reuse existing C++ libraries. QtWebEngine also offers an option to display HTML/JavaScript code together/instead of the QML, while offering a similar mechanism of communication with C++ code using the QtWebChannel. /15/

2.5 Server-Client communication

As one of the main requirements of the Smart Client application is exchanging data with the Gateway this chapter will explore possible solutions for the client-server communication supported in the Qt framework.

A very popular technique used in web applications is AJAX, which uses a JavaScript XMLHTTPRequest (supported by most of the modern web browsers according to the W3 specification /16/) to continuously request new data from the server without having to refresh the whole website. As the QML JavaScript host environment implements the XMLHTTPRequest /17/, the AJAX communication can be build using JavaScript directly from the QML code in a very similar fashion.

One of the AJAX alternatives that became popular in the recent years are the WebSockets which, contrary to the request/response type of communication offered by AJAX, provide a continuous, low-latency and bi-directional communication with the server. Qt supports WebSockets with by offering a QML type of the same name available in the QtQuick library /18/.

Of course a multitude of other solutions is available outside of the QtQuick by using any of the C++ (and not only) libraries implementing web service protocol like SOAP or XML-RPC.

In theory, the Apache Thrift framework can be used to generate interface of the services provided by the server as JavaScript code to be used directly from QML without writing any C++ extension. /19/

2.6 Development environment

The Qt Company offers own Integrated Development Environment (IDE) along with Qt framework under name Qt Creator which is not only a sophisticated code

editor for C++ and QML but also provides version using version control system like GIT or Subversion and build management. /20/

Especially useful in cross-platform development is also support for deploying applications on iOS mobile device, Android device via Android Debug Bridge (ADB) or remotely on Linux device via SSH. It's also possible to use iOS and Android emulators directly from Qt Creator.

Although it's possible to successfully use Qt framework from other IDEs like CLion or Visual Studio the Qt Creator is the only major IDE at the moment that provides support for editing QML code and designer for QML/QWidgets.

Toolchain necessary for building applications comes together with Qt installer for all officially supported architectures.

2.7 Alternative solutions

This thesis focuses primarily on exploring Qt as the framework as a tool for the cross-platform development; however, it's worth comparing it to the other alternative solutions currently being popular on the market.

PhoneGap - similarly as Qt provides own WebView deployed together with HTML/JavaScript code on the client device and can be extended by custom functionalities. Support only mobile platforms.

Xamarin - uses C# language and Mono .NET framework having Microsoft behind it. C# code is compiled to a native or intermediate (IL) language depending on the platform. Uses native user controls preserving platform look and feel and supports both mobile and desktop platforms.

Haxe - a framework that offers own programming language that can be source-to-source compiled to other languages supported by the target platforms. In addition, Haxe toolkit comes with set of libraries like HaxeUI - for creating the user interface. Supports all major desktop and mobile platforms.

Java - using JavaFX to develop user interface it's possible to create applications for all major desktop platforms. Mobile devices are not officially supported; however, it is possible to deploy application on iOS and Android using JavaFXPorts by Gluon. Downside of Java as cross-platform solution is that client requires Java Runtime Environment which cannot be deployed together with the application.

2.8 Summary

Information gathered in during this chapter has shown that Qt framework should be capable of satisfying all major Smart Meter's requirements as the software development tool.

The following table summarizes the design choices based on this study that will be used in the prototype implementation and the practical assessment of the Qt framework.

Table 1 Summary of the design choices for the prototype implementation.

Concept	Choice and justification
Client archetype	The Hybrid Application. Both, Rich Client Application and Rich Internet Application archetypes can be built with Qt. However, the fact the Smart

	Client will operate on many devices, with different screen sizes and screen types, makes The Hybrid Application the best choice.
GUI technology	QML with QtQuick QML seems to be the only way to implement The Hybrid Application. This declarative language can be used to implement different GUI version for each platform, in form of QML files that could be fetched remotely from the Gateway device. In addition, the documentation and online sources suggest that QWidgets are meant to be used with desktop devices. Indeed, QtQuick offers more modern controls/widgets from both: visual side and technical side with gesture and animation support.
Native access	QML C++ extensions C++ extensions provide an easy way to expose the external libraries, written using the native SDK, to the QML views.
Development tools	Qt Creator Qt Creator IDE should provide all tools necessary in the prototype development without need to access any third party software.
Client-server communication	WebSocket Both: XMLHTTPRequest and WebSocket are viable tech-

technology	nologies that meet all requirements and are supported by Qt libraries. WebSocket has been chosen as the implementation of the bi-directional communication between Client and Gateway will be easier.

3 CLIENT PROTOTYPE

The main goal of the implementation phase of this project is to provide a proof of concept, showing that Qt framework can be successfully used in the development of a modern, cross-platform application. After presenting the general design of the prototype application, the consecutive chapters discuss in detail the implementation of the requirements' goals, defined in the statement of the problem.

The source code of the application can be found in appendices.

3.1 Design

Diagram below depicts the basic architecture of Client, Gateway and interaction between them:

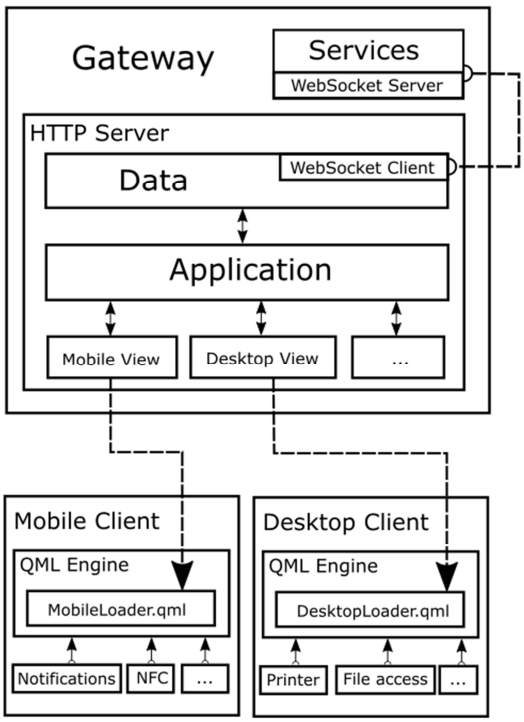

Figure 3 Overview of the prototype design

Client:

The client device requires a minimalistic sandbox that uses a different Qt runtime environment depending on the host platform. The Client's code itself is fairly trivial: it has to start the QML engine, register the optional C++ extensions like the push notification or camera control, and provide the QML loader file. Depending on the platform/device, it will load the different View from the Gateway. Once the View is loaded, the application is controlled completely by the QML code that, when needed, uses the C++ extensions provided by the Client.

Gateway:

The Gateway has two responsibilities regarding the Client: serving the QML files over the network and providing various services according to the business logic. In the implemented prototype those services consist of:

- Information about devices present in the power grid (their ID, power consumption).
- Information from the electricity provider like the current power price.
- Handling the requests to control those devices (changing their parameters: turning ON/OFF, updating thumbnail and others)

QML files are available to the Client through the HTTP server; whereas the services are available by communicating with the WebSocket server implemented using Tyrus library.

3.2 Remote QML Loading

All QML files used by the client application are stored on the Gateway device. The only exception here is the Loader.qml, with the only purpose to display appropriate error to the user when the connection with Gateway cannot be open. It should be noted that this way of implementing application is discouraged as it might considerably hinder its performance. Nevertheless, the application prototype is simple enough to provide a responsive GUI.

QML files are stored on the Gateway in a simple folder structure:

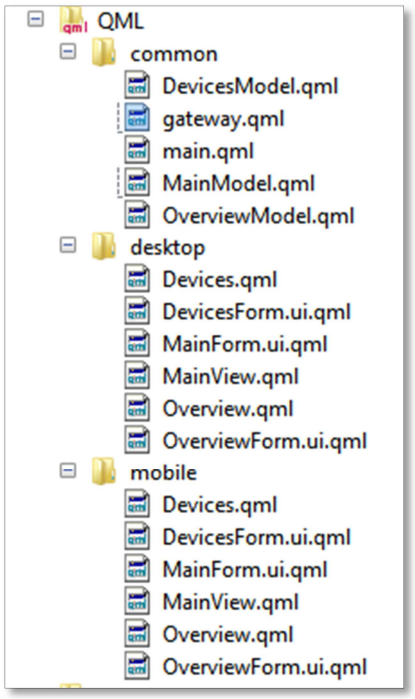

Figure 4 Folder structure for QML files on the Gateway

The QML loading starts when Loader.qml requests the main.qml file from the Gateway. This is done by using the QML type Loader.

```
Loader {
        source: "http://192.168.1.2:9000/common/main.qml"
}
```

Once the main.qml is loaded it can refer to other QML files as if they were on the same, local folder. This makes all QML files located on the Gateway unaware of the fact they are loaded over the network.

During the development of the application prototype, the Gateway file sharing was handled by using the Mongoose web server running on a PC machine.

The main.qml file then passes control to the MainModel.qml which initializes all views from desktop or mobile directory, depending on the client device type. To

the each module is assigned an appropriate model. Models are shared between all Views implementations, hence, are located in the common directory.

3.3 Gateway-Client communication

Data requests made by the clients are implemented using WebSockets. During the prototype development, the Java Tyrus WebSocket server was used to emulate a working Gateway. On the client side, the Gateway.qml is responsible for maintaining the connection using the WebSocket QML type. There are three types of requests implemented in the prototype application used to implement the Gateway-Client communication:

- Single request – allows client to request a single piece of information or change of certain parameter of the system.
- Periodic request – provides a continuous data exchange allowing the application views to keep their values updated.
- Gateway request – a request initiated by the Gateway used to display any kind of notification or alarms to the user.

It should be noted that all request type described in this chapter are executed asynchronously, which keeps the user interface responsive even when waiting for the Gateway response.

3.3.1 Single request

The following diagram illustrates details of processing a single request from the Client to Gateway that changes a device's parameter:

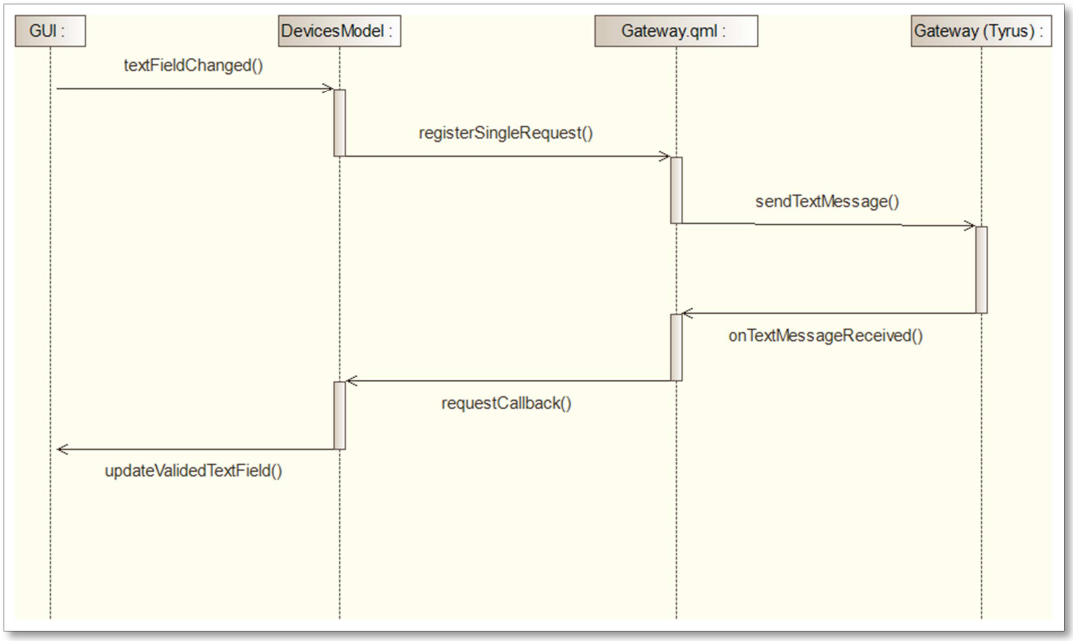

Figure 5 Sequence diagram of executing the single request

In this example, user modifies a particular parameter of the device by changing value of a text field visible in the GUI. Once change is detected by the view model, it registers the request to the Gateway.qml. The request consists of two elements:

- ID of the request – example: a string "updateDeviceName"
- Additional request parameters – example: ID of the device and new name

The Gateway.qml sends the registered request to the Gateway device which is processed and returned with appropriate data. Request callback, provided by the model, is then called allowing to print information to the user that name has been successfully changed or print a validation error.

3.3.2 Periodic request

By registering a periodic request, the view model doesn't have to maintain any kind timer triggering data update. Instead, this responsibility is passed to the Gateway.qml. Once registered, the requests are periodically repeated until model unregisters them.

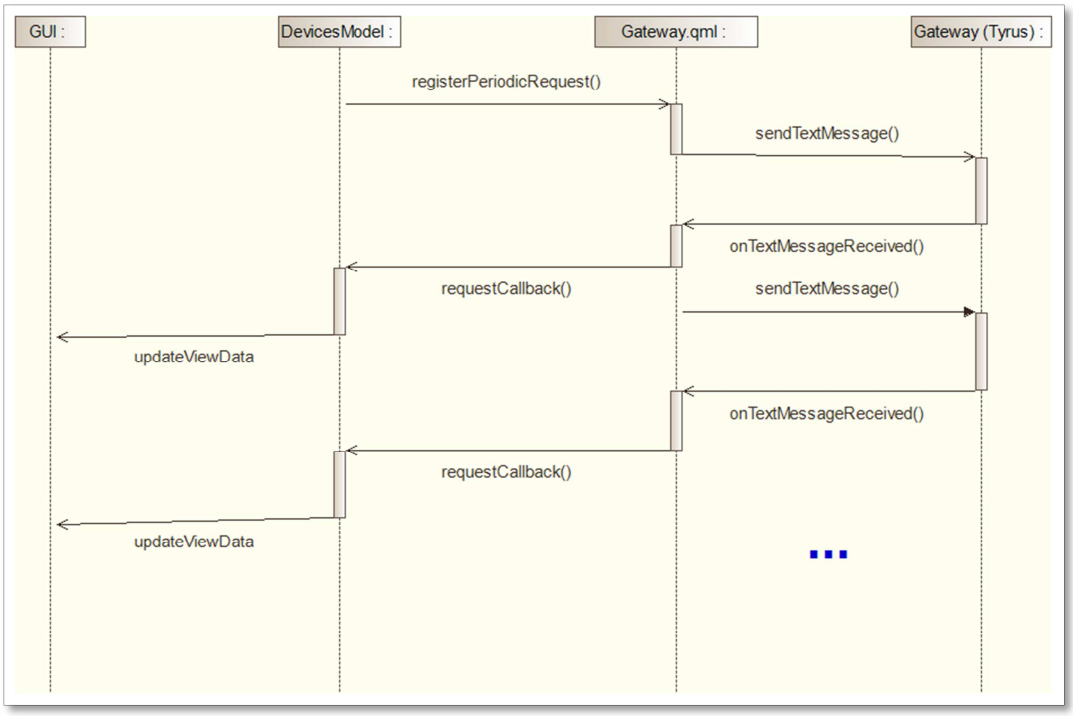

Figure 6 Sequence diagram of executing the periodic request

3.3.3 Gateway requests

Since the WebSockets connection between Client and Gateway is maintained throughout the whole application's lifecycle, it's possible to support the bidirectional communication:

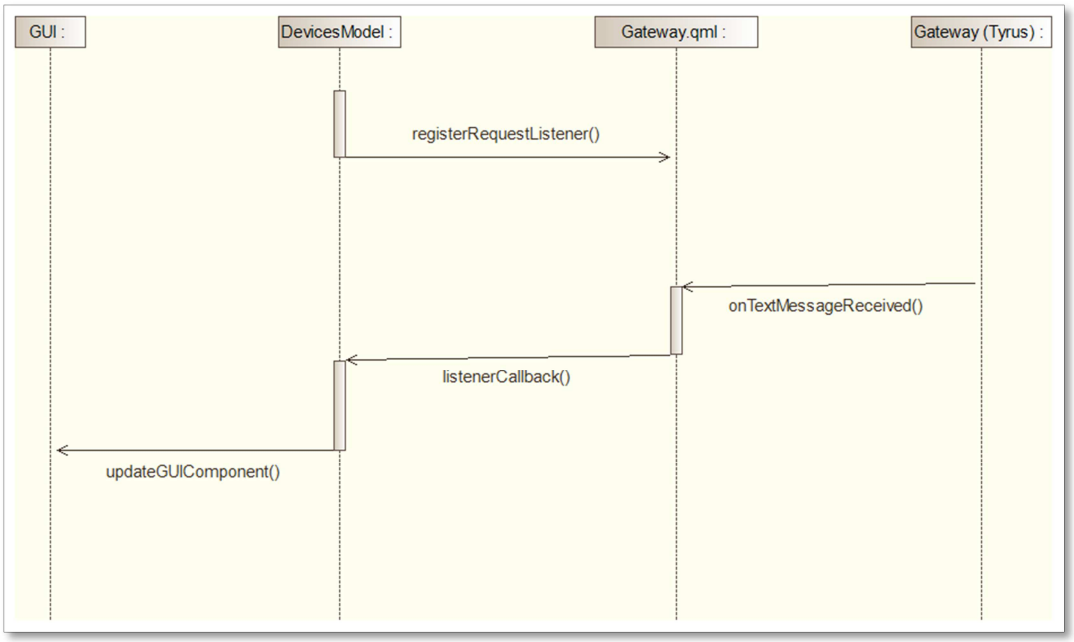

Figure 7 Sequence diagram of handling the request orignated from the Gateway

The view model can subscribe itself to the incoming Gateway requests by registers a request listener. Whenever Gateway.qml receives a matching request the registered callback is invoked.

3.4 Graphical User Interface

The Smart client prototype contains a simple Graphical User Interface that consists of the following views:

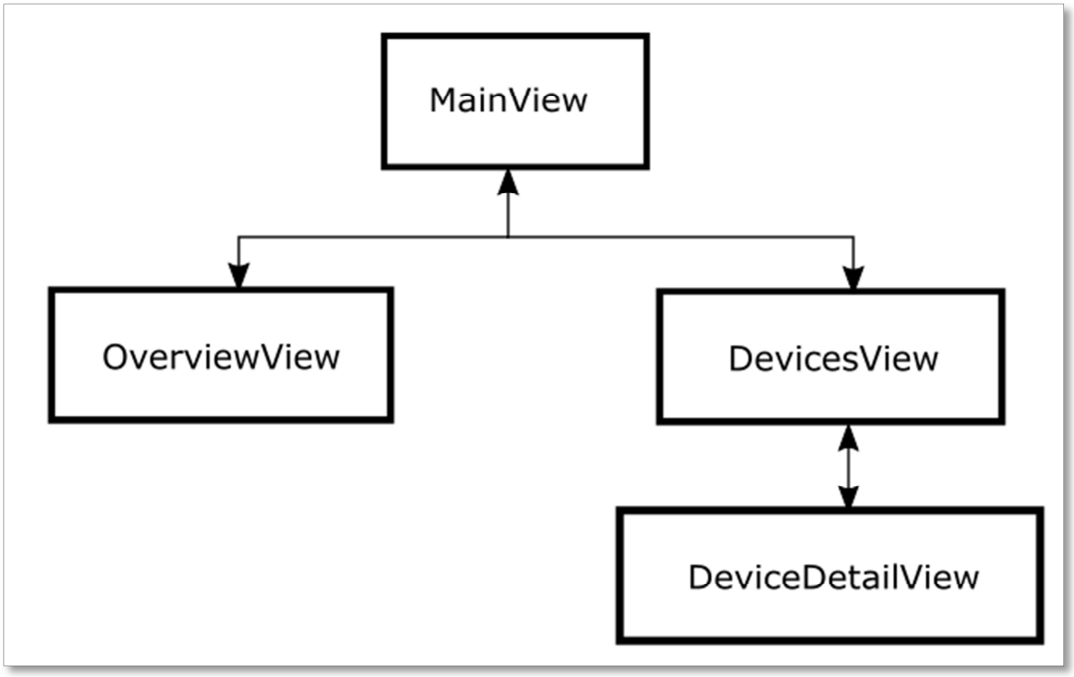

Figure 8 Views hierarchy

MainView contains the top-most controls like MenuBar, Toolbar or SwipeMenu; which allow for navigating between the rest of the views. OverviewView is the default view, set on the start of the application. It contains a few of the QtChart controls visualizing statistics of the power network like power consumption and the monthly electricity price. DevicesView lists all devices connected to the power network and allows for changing device's parameters through the DeviceDetailView. Although relatively simple, this design is sufficient to demonstrate capabilities of Qt framework in building a modern Graphical User Interface.

As described in the previous chapter, to provide a usable GUI on devices with different operating system and screen type, the prototype application contains two implementations of the View layer:

- Desktop – for personal computer using a desktop operating system and an external monitor or other large display without touchscreen.
- Mobile – for mobile device with touchscreen display of size 4-6 inches

All screenshots presented in this chapter come from one the following devices used in the testing:

- PC, Windows 8.1
- PC, Linux Mint 17.3 Cinnamon
- Nexus 10 tablet, Android 5.0, 10-inch touchscreen display
- OnePlus One Smartphone, Android (Cyanogen) 5.1.1, 5,5-inch touchscreen display.

3.4.1 Views navigation

The main responsibility of the MainView is to provide navigation between other views of the application. The desktop implementation of the view uses a TabView control to do it, where each tab contains different view available to the user:

Figure 9 Screenshot of the MainView - Windows

Implementation for mobile device uses animated swipe menu. It acts as on overlay, invisible when not used. Once a swipe gesture is detected it makes menu visible on the left side of the screen. Pressing the menu button changes the view underneath and hides the menu.

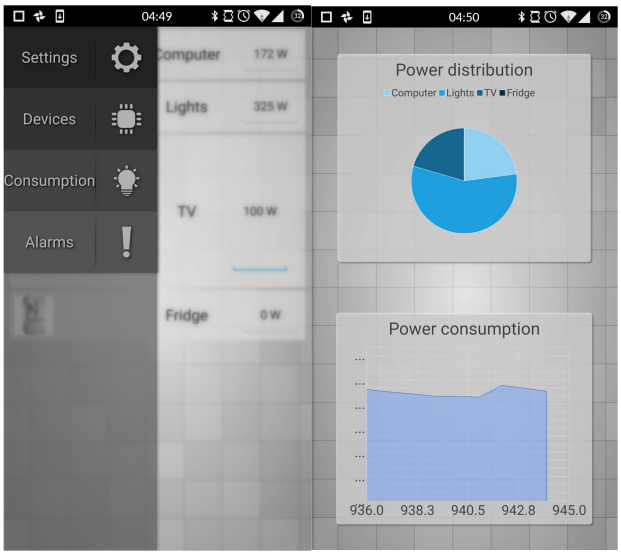

Figure 10 Swipe menu - Smartphone

3.4.2 Screen orientation

Most of the Smartphones and tablets currently available on the market allow user to change screen orientation. Three ways were considered to support this feature in the prototype's GUI:

- Allow only a single screen orientation – application will maintain the same orientation independently from the device's screen orientation. On Android this can be done by modifying application manifest file.
- QML states – each orientation will have a QML state assigned to it. Once orientation changes, so will its state. Then, by manipulating various layout parameters, like anchors, state can adjust the view according to its orientation.
- Separate view implementation – For the most complex cases, where manipulating the layout parameters using states is too complicated to achieve desired effect, each screen orientation can have a separate implementation of the view.

The Smart Client prototype uses using states to support both screen orientations:

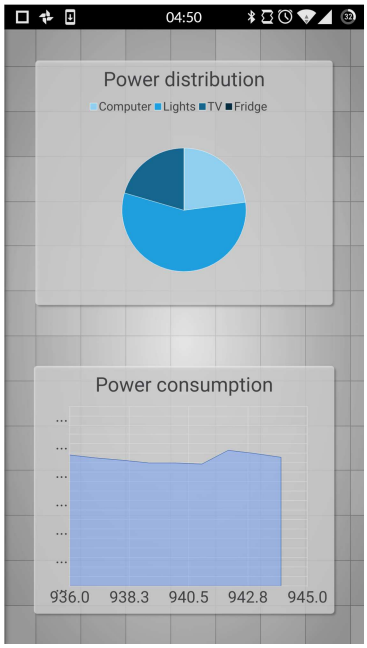

Figure 11 Portrait orientation - Smartphone

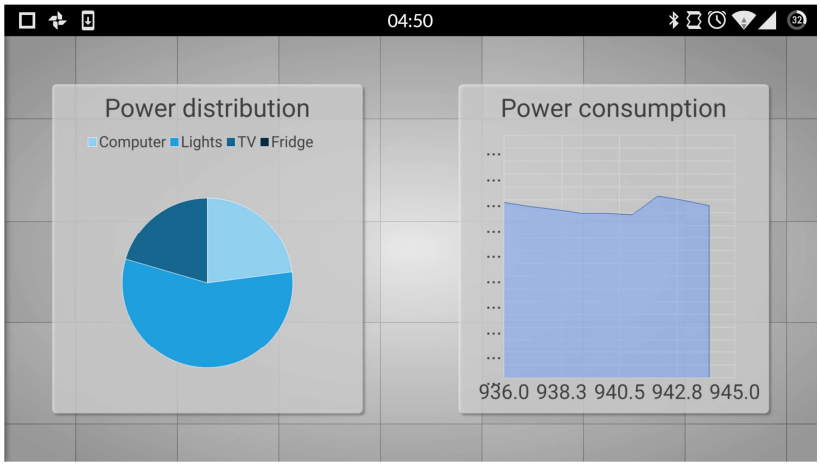

Figure 12 Landscape orientation - Smartphone

3.4.3 Native controls

The controls available through the QtQuickControlls and QtQuickControlls2 modules offer the look and feel that is similar to the one used by the operating system on the target device. For example, the MenuBar control will look and behave differently depending whether it is run on Windows or Android. Following screenshots of the prototype application demonstrates the application window with the MenuBar and Toolbar controls defined by the same QML file:

Figure 13 MenuBar, Toolbar and TabView controls on Windows

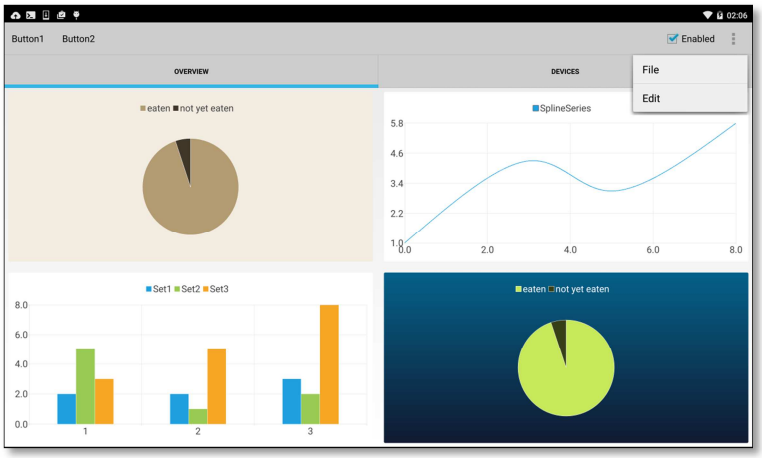

Figure 14 MenuBar, Toolbar and TabView controls on Android tablet

Screen shots above show that the same the QML control can provide the user experience very similar to the one expected on the operating system it is run on.

3.5 Cross compilation

3.5.1 Toolchain

There are two approaches to develop an application on the Raspberry Pi device: directly on the device itself or on another machine using the cross compilation toolchain.

At the moment of writing this report, the latest version of Qt available on Rasbian software repository is 5.3.2. Hence, the Qt framework had to be compiled from the available source code to match the version used on other platforms: 5.7. Taking into account the required disk space and the compilation time, the Qt source code was compiled on the x86 Linux machine. The side effect of this task is building a complete Qt toolchain allowing for cross compiling Qt applications for Raspberry PI.

Building Qt framework was done based on the official Qt guide /7/. Following components were used:

- Raspberry Pi Tools containing the arm-bcm2708-linux-gnueai GCC cross-compiler allowing for compiling code for ARMv7/v6 (depending on Raspberry PI version) on a Linux-x86 host. /22/
- Raspbian Jessie /23/ – Debian-based operating system recommended by the RPI manufacturer.
- Sysroot – copy of all required libraries and headers obtained from the target device and used to configure and build Qt framework.
- Qt source code /24/
- Raspberry Pi v1. device

First step was to build the qmake and other tools that, together with GCC compiler, form a complete Qt cross compile toolchain:

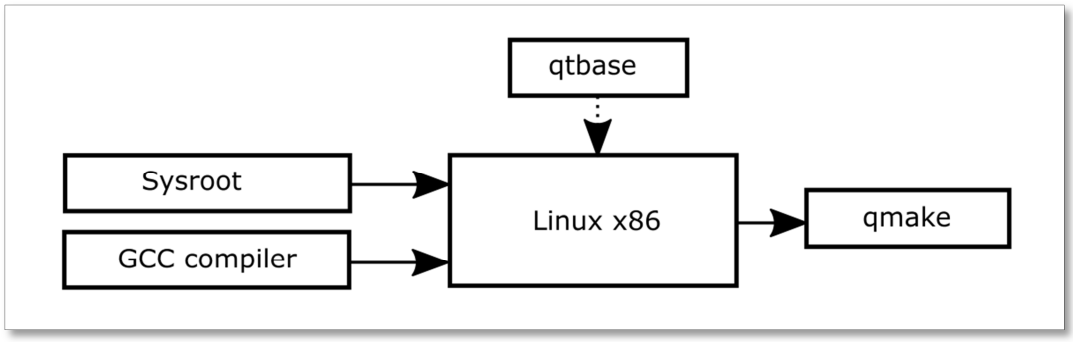

Figure 15 Using the Raspberry Pi cross compiler to build qmake.

Qt source code is divided into modules. The Qtbase module contains the most essential features and tools like qmake allowing for building the Qt applications. Once Qtbase tools have been compiled all other required modules were built. For example, Qtdeclarative module, which contains QtQuick library used to develop most of the application's code:

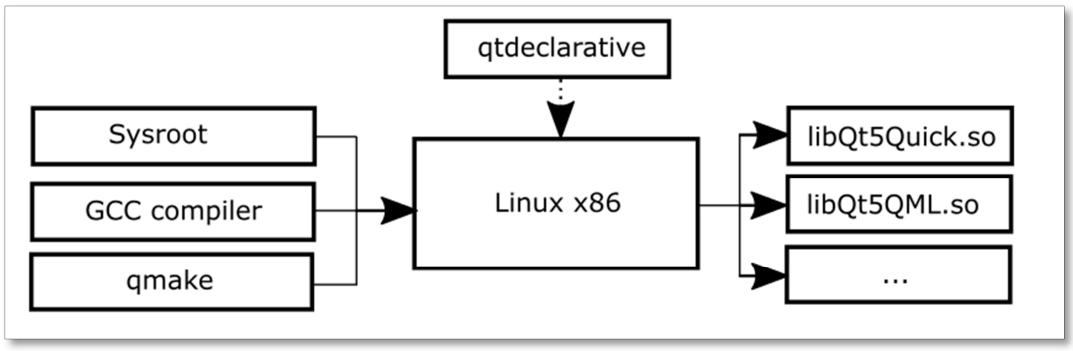

Figure 16 Building other optional Qt libraries

Once copied back to the Raspberry Pi device, the shared libraries can be loaded and used by any Qt application during runtime.

3.5.2 Qt Kit

Qt Creator allows for organizing the available toolchains in so called Kits. Following snapshot shows a Kit configuration dialog with settings used to build the Smart Client prototype for the Raspberry Pi.

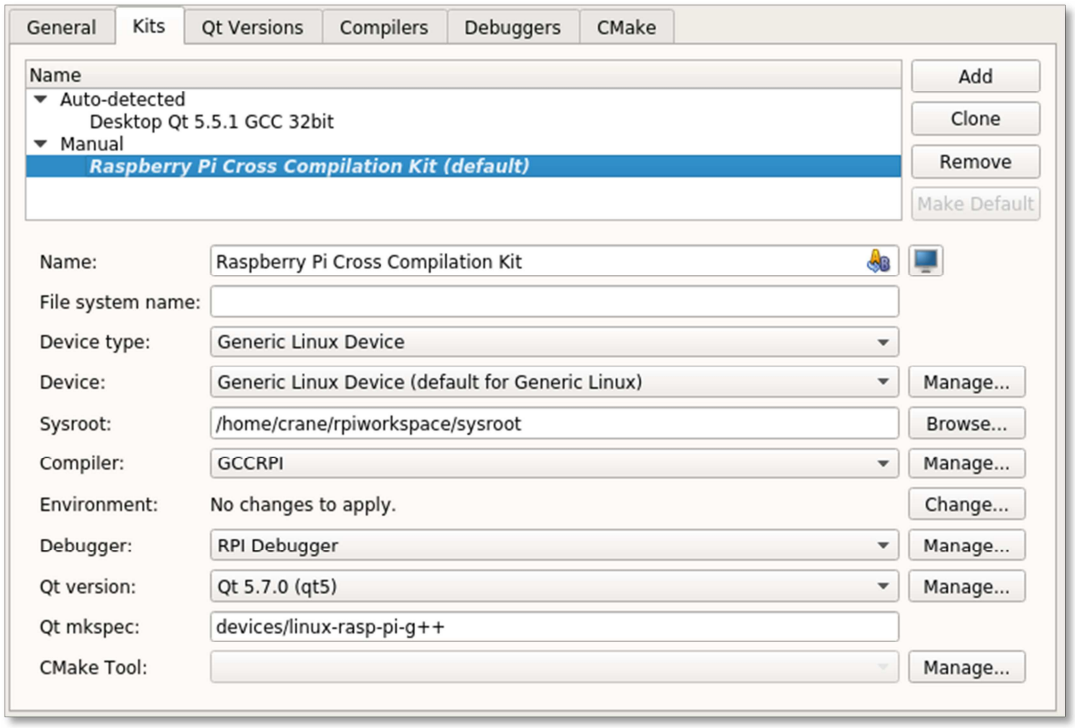

Figure 17 Configuration of the cross-compile Raspberry Pi kit in the Qt Creator

Each Kit consists of the following elements:

- Sysroot – root directory of the target device containing headers and libraries used during the build.
- Debugger
- Compiler
- Qt version – location qmake and other Qt configuration files
- Qt mkspec – additional compiler settings for the target platform/device

3.5.3 Remote deployment

An additional feature of the Qt Creator is the automatic deployment and execution of the built application over SSH. Following picture shows settings used to define the Raspberry Pi as a target device.

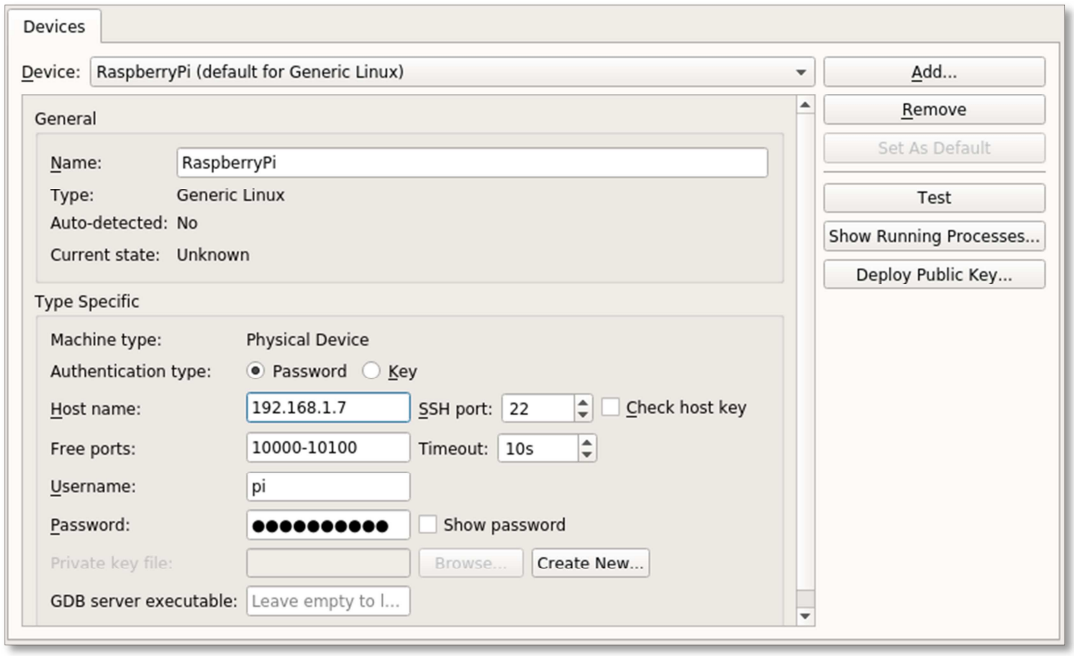

Figure 18 Configuration of Raspberry Pi as a remote test device in Qt Creator

With the Qt Kits fully defined, it's possible to quickly switch between running application of the host system or remote device:

Figure 19 Panel allowing to quickly switch Kit used in the current build.

When run remotely, Qt Creator will open SSH connection with specified device, deploy the binaries to the remote directory specified in the pro file, and start their execution with standard input/output available in the Qt Creator's console.

3.6 Additional tools

3.6.1 Debugger

As any modern Integrated Development Environment, the Qt Creator offers a build-in debugger including features like:

- conditional breakpoint
- stack trace
- monitor of the scope variables
- Instruction-wise mode for debugging assembly code

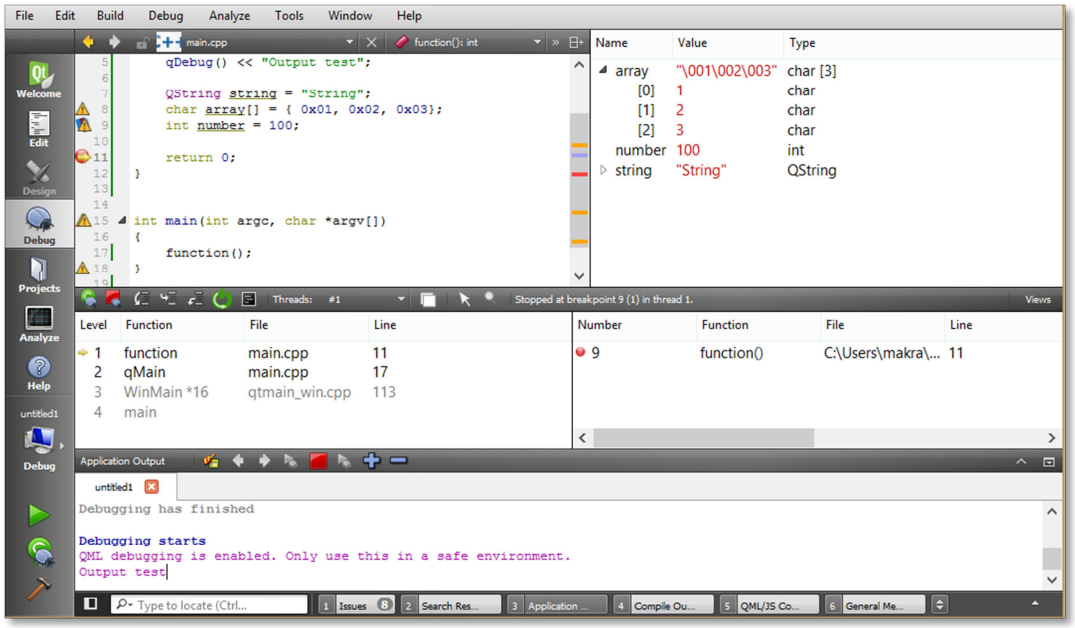

Figure 20 Debugger in the Qt Creator.

The debugger has been used extensively during the prototype development. Especially useful was the ability to debug the QML code and remote debugging the application running on Android or Raspberry Pi.

3.6.2 QML Designer

An alternative to develop the Graphical User Interface by manually editing QML files is to use the QML Designer, which is bundled with Qt Creator installation.

While editing any QML file it is possible to switch to the design mode which opens the QML Designer view and allows for the graphical editing. Like most of the popular UI tools it offers drag and drop style editor combined with settings of the various properties: connecting signals, binding and dynamic properties, and properties of the QML controls/widgets themselves.

Although QML designer is capable of editing any QML file, the official guide (citation needed) recommends using the provided wizard that splits a view into two QML files:

- ${FileName}Form.ui.qml – which contains pure QML code without any logic written in JavaScript
- ${FileName}.qml – which binds to the signals exposed by the Form.ui file, sets its properties and implements simple JavaScript logic

The SmartClient prototype uses a similar separation when implementing its views. For example, the Devices view, which contains list of the devices detected in the power network, consists of DevicesForm.ui.qml and Devices.qml files.

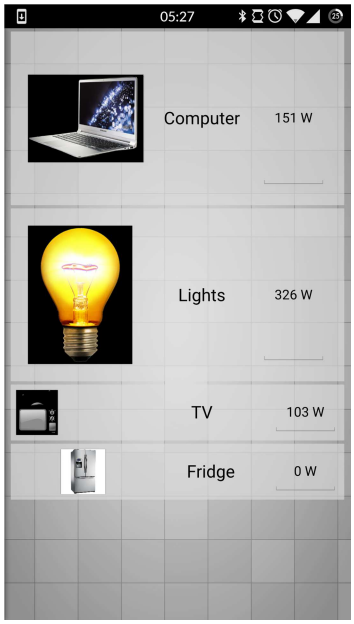

Figure 21 Screenshot of the Devices view

The DevicesForm.ui.qml file describes the layout of the QML controls in the view, for instance the ListView; whereas, the Devices.qml file handles signals triggered by the pressed buttons and fills the ListView with the model's data.

The QML Designer has been successfully used throughout the project, especially for the initial design of the views. However, some of the more complex views containing controls like Layouts and Charts could not be processed in the design mode. In those cases, the design mode was not available, and files had to be edited manually. It should be noted, that at least part of the encountered issues could be caused by using the latest Qt 5.7 beta release and fact that it is the first version that introduced controls like Charts.

3.7 Push notifications

The push notification is a technique of displaying a notification to the user even if application is running in the background. It is commonly used on the mobile devices. It was implemented in the prototype by integrating example available in the Qt documentation /21/ and demonstrates two important mechanisms:

- Creating a C++ extension that can be registered in the QML engine and used as a QML type.
- Using the Android's native API with Java code without the direct JNI calls (the JNI calls are covered in the camera control implementation).

As demonstrated on the sequence diagram below, QML application calls athe function of the PushNotification C++ class whenever notification should be displayed. PushNotification object is registered in the QML context as a singleton, i.e. only a single its instance is available. Then, the Java Android activity is run, which uses directly the Android API to display the notification.

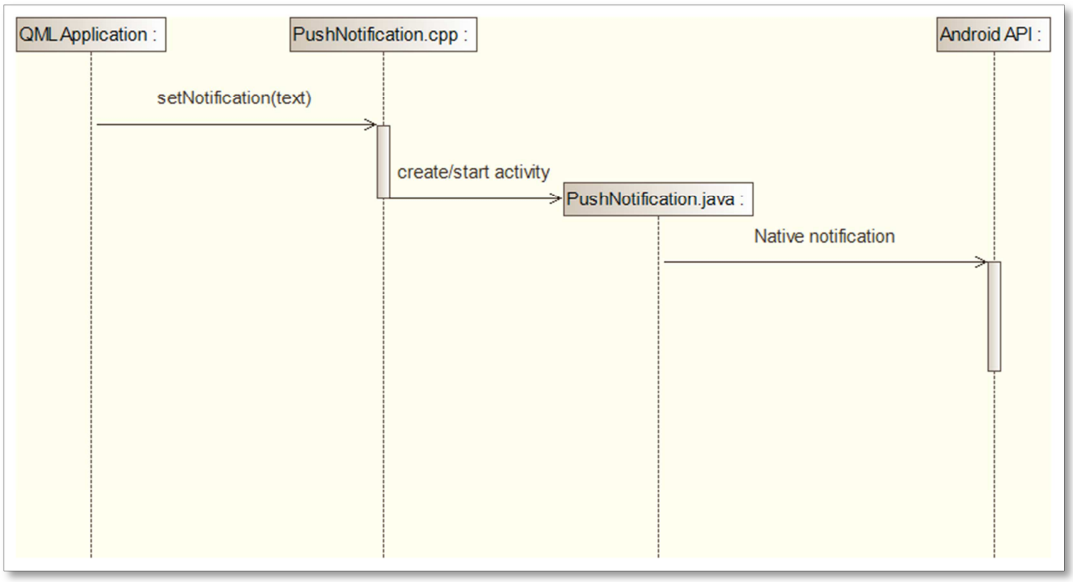

Figure 22 Sequence diagram of the Push Notification extension

The main benefit of this solution is that complex activities can be written entirely in Java and included as additional files in the project *.pro file. Qt Creator is then able to bundle them together with other C++ files and deploy on android device.

3.8 Camera control

Implementation of the camera control is done through the Android SDK library to demonstrate how QtQuick application can access the native environment of the client. Otherwise, this task could be accomplished through the available Camera QML type with much less effort. Camera control C++ extension is exposed to the QML as ThumbnailSnapper type. Invoking its function: snapThumbnail(), opens the default Android camera application and allows user to take a single picture that is then set as the device's thumbnail.

Contrary to the push notifications feature, the camera control is implemented without any Java code, purely through the JNI mechanism offered by Qt.

The sequence diagram of the ThumbnailSnapper is following:

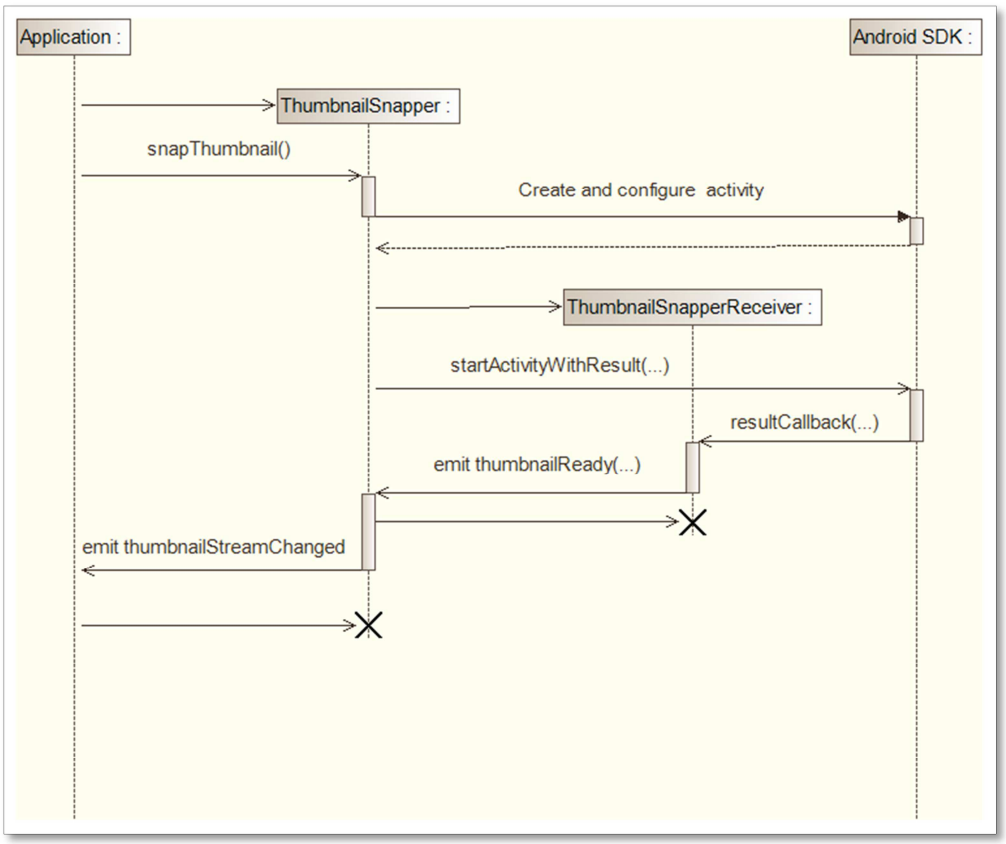

Figure 23 Sequence diagram of the ThumbnailSnapper extension.

3.8.1 Native Java implementation

Implementation started with writing camera control code in Java using Android Studio and Android SDK. It turned out to be a quite efficient way of writing further JNI code in Qt. Since the QAndroidJniObject class requires providing a signature of the method (as QString) being invoked, it's prone to typos which are detected only during the application execution on the device or emulator.

```
Intent intent = new Intent();
intent.setAction("android.media.action.IMAGE_CAPTURE");
File photoFile = new File(Environment.getExternalStorageDirectory(),
"Photo.png");
imageUri = Uri.parse(photoFile.toURI().toString());
intent.putExtra(MediaStore.EXTRA_OUTPUT, imageUri);

int CAPTURE_IMAGE_ID = 1234;
startActivityForResult(intent, CAPTURE_IMAGE_ID);
```

This code, assigned to a button press event, creates a new Intent with IMAGE_CAPTURE action. Once started, it opens a default Android camera application and allows user to take a single picture. The MediaStore.EXTRA_OUTPUT parameter controls the output file to which the final picture should be saved to. Once Intent object is fully set the activity is started.

3.8.2 Qt implementation

After the activity written in Java has been successfully tested on Smartphone device, the code was ported to the Qt application.

The equivalent C++ code is following:

```
QAndroidJniObject mediaDir = QAndroidJniObject::callStaticObjectMethod("android/os/Environment",
"getExternalStorageDirectory", "()Ljava/io/File;");
QAndroidJniObject mediaPath = mediaDir.callObjectMethod( "getAbsolutePath",
"()Ljava/lang/String;" );
QString picturePath = "file://" + mediaPath.toString() + "/smartClientImage.png";

QAndroidJniObject uri = QAndroidJniObject::callStaticObjectMethod("android/net/Uri", "parse",
"(Ljava/lang/String;)Landroid/net/Uri;", QAndroidJniObject::fromString(picturePath).object<jstring>());
QAndroidJniObject action = QAndroidJniObject::fromString("android.media.action.IMAGE_CAPTURE");
QAndroidJniObject intent("android/content/Intent");

QAndroidJniObject extraOutput =
QAndroidJniObject::getStaticObjectField("android.provider.MediaStore", "EXTRA_OUTPUT",
"Ljava/lang/String;");
intent.callObjectMethod("setAction", "(Ljava/lang/String;)Landroid/content/Intent;",
action.object<jstring>());
intent.callObjectMethod("putExtra",
"(Ljava/lang/String;Landroid/os/Parcelable;)Landroid/content/Intent;",
extraOutput.object<jstring>(), uri.object<jobject>());
 connect(receiver, &ThumbnailSnapperReceiver::thumbnailReady, this, &ThumbnailSnapper::handleReadyThumbnail);

QtAndroid::startActivity(intent, actionCode, (ThumbnailSnapperReceiver *)receiver);
```

Qt allows for calling the native Java API leveraging JNI – Java Native Interface called through helper class QAndroidJniObject.

The QAndroidJniObject itself is representation of a Java class instance: from primitive types like int or float to complex classes like java.io.File. It offers number of methods allowing for instantiating a new class object and execute its methods.

For example, the line:

```
QAndroidJniObject intent("android/content/Intent");
```

initializes the class Intent under package android.content with constructor taking no arguments. Such initialized variable intent, now holds the Intent class instance and is further used to call the Java object method “putExtra”:

```
intent.callObjectMethod("putExtra",
"(Ljava/lang/String;Landroid/os/Parcelable;)Landroid/content/Intent;",
extraOutput.object<jstring>(), uri.object<jobject>());
```

The second argument of the callObjectMethod function describes the exact signature of putExtra method. This mechanism allows for calling the overloaded methods using the same name. The signature must be written using following syntax: (argumentType1, argument2Type, …)returnType. Hence, the signature shown above represents the following Java method prototype:

```
android.content.Intent putExtra( java.lang.String, android.os.Parcelable);
```

QAbstractJniObject offers similar mechanisms for invoking static methods of the class and also accessing static/non-static field of an object.

The last line of the code starts the activity and provides a callback function (through the interface QAndroidActivityResultReceiver) which should be invoked once activity execution is finished. The definition of the callback function is following:

```
void ThumbnailSnapperReceiver::handleActivityResult(int receiverRequestCode, int resultCode, const
QAndroidJniObject &data) {
    jint RESULT_OK = QAndroidJniObject::getStaticField<jint>("android/app/Activity",
"RESULT_OK");
    if (receiverRequestCode == actionCode && resultCode == RESULT_OK) {
        QFile file("/storage/emulated/0/smartClientImage.png");
        file.open(QIODevice::ReadOnly);
        QByteArray byteArray = file.readAll();

        emit  thumbnailReady(byteArray.toBase64());
    }
}
```

Prototype of this function is very similar to the native activity onActvitiyResult callback:

```
void onActivityResult(int requestCode, int resultCode, Intent data)
```

It performs a simple check validating the result and copies the content of the picture file into the QString, using Base64 encoding. Encoded picture is then emitted as a signal and eventually set as the property of the ThumbnailSnapper QML object.

4 SUMMARY

Cross-platform support:

- According to the research, the hybrid application has been chosen as the most feasible application type for the given requirements.
- According to the documentation Qt framework can support development for all required platforms: Windows, Linux, Android, Mac OS X and iOS
- Prototype application has been successfully deployed on required platforms: Windows, Android and Raspberry Pi running Raspbian Linux.

Graphical User Interface:

- According to the documentation, Qt framework is a suitable framework for GUI development, offering a wide variety of widgets and controls via QtQuick module.
- Proven that QML language allows for developing a highly customizable, responsive and animated user interface.
- Prototype application successfully implemented GUI suitable for both Smartphone and Desktop devices which:
    - Uses QML controls to represent: charts, lists, menu bars, toolbars and others.
    - Demonstrates how animation and gestures usage by implementing swipe menu
    - With limited set of controls provides the native look and feel of the operating system it is run on.
    - Is responsive.
    - Supports portrait and landscape screen orientation when used on the Android device.

Client-Server Communication:

- WebSocket chosen as the technology most suitable for implementing the client-server communication in the Smart Client.
- Implemented a Gateway emulator able to provide data to the client with WebSocket.
- Implemented the WebSocket communication in the application prototype demonstrating usage of all request types:
    - Single client request
    - Periodic client request
    - Server request

Native access of the client device:

- Implemented the QML C++ extension plugin that is able to send the push notifications on the Android device and demonstrates the Android API access using Java code.
- Implemented the QML C++ extension plugin that is able to take a picture using camera available on the Android device and demonstrates the Android API access using JNI API provided by Qt framework.

Development tools:

- Built a Linux cross-compile toolchain able to compile Qt applications for Raspberry Pi device.
- Built the Qt runtime for Raspberry Pi device.
- Qt Creator was used through the prototype development and assessed.
- Qt Creator debugger was used through the prototype development and assessed.
- QML designer was used through the prototype development and assessed.

5 CONCLUSIONS

The researched information have been successfully applied in practice and demonstrated that Qt framework can be successfully used to develop cross-platform applications on Android, Windows and Linux operating systems. Study was limited to just those three platforms; however, The Qt Company declares support also for a few others: Windows RT, Mac OS X and iOS. This makes Qt framework one of the very few solutions available on the market which supports such a wide number of operating systems and architectures.

Furthermore, the Qt sources were used to successfully build a cross-compile tool-chain for the Raspberry Pi device. This is especially important for the Smart Client project giving possibility to develop the HMI client on virtually any device using a Linux system.

Tools offered by The Qt Company are of high quality. The Qt creator can rival with other major IDEs available for the C++ language. It also offers assistance when deploying and debugging application remotely, which turned about to be invaluable during the development for Android and Raspberry Pi devices. The only tool that did not quite meet the expectations is the QML designer. Although it looked very promising at the beginning, in did not work with all controls and widgets used in the application, which limited its usage considerably.

Another Qt downside is the deployment time on the Android device, which comparing to the Android Studio, was roughly half a minute longer. This might not seem like much, but considering the rather typo-prone JNI API, it makes debugging the Android application a tedious task.

QML was proven to be a powerful and flexible language allowing for developing a highly customizable user interface using an extensive library of QtQuick controls. Style of the most controls can be further customized or, if set to default, used to simulate look and feel of the target operating system.

QML can be further extended by custom QML types written in C++. This let developers use other 3rd party libraries in their project or, using JNI API, call Android native libraries.

Finally, the quality of documentation offered by The Qt Company is positively surprising. It contains extensive examples and very descriptive, up-to-date API wiki pages available at any moment through the Qt Creator help mode or web site.

In retrospective, the topics like client-server communication and implementation of the Gateway emulator were not exactly crucial to the outcome of this thesis. Time spent on their research and implementation could be used better to investigate more important issues. This evaluation would benefit greatly from better insight into the Qt alternatives currently available on the market. Even though Qt seems to satisfy every crucial requirement of the Smart Client, it’s impossible to make a decisive verdict without putting it in the comparison with the other popular frameworks like PhoneGap or Xamarin. Other important topic that should be discussed is the capability of the Qt framework to test the application that is being developed. It does offer tools like QtTest module (for C++ code) and TestCase QML type (for QML code) which should be assessed together with other frameworks like the Squish GUI Tester.

6 REFERENCES

/1/ U.S. Department of Energy. The Smart Grid. Accessed 18.5.2016. https://www.smartgrid.gov/the_smart_grid/smart_grid.html

/2/ U.S. Department of Energy. The Smart Home. Accessed 18.5.2016. https://www.smartgrid.gov/the_smart_grid/smart_home.html

/3/ IEEE. Smart Grid. 2016. http://smartgrid.ieee.org/

/4/ International Energy Agency. 2016 https://www.iea.org/topics/electricity/subtopics/smartgrids/

/5/ Alessio Montone, Network Digitalisation, Enel Point of View, 24.11.2015. http://www.smartgrids.eu/documents/eventsandworkshops/2015/7_ETP_SmartGrids_Workshop_24th_November_2015_Alessio_Montone.pdf

/6/ The Qt Company. Qt Licensing. Documentation for Qt 5.6. http://doc.Qt.io/Qt-5/licensing.html

/7/ The Qt Company. RaspberryPi Beginners Guide. Qt Wiki page. https://wiki.Qt.io/RaspberryPi_Beginners_Guide

/8/ Microsoft Application Architecture Guide, Choosing an Application Type. 2nd Edition2. October 2009. https://msdn.microsoft.com/en-us/library/ee658104.aspx

/9/ Sten Anderson. The Advantage of Hybrid Clients in Enterprise Applications. 2010. http://www.citytechinc.com/content/dam/citytechinc/pdf/Hybrid-Clients-in-Enterprise-Apps2.pdf

/10/ Henning Heitkotter, Sebastian Hanschke, Tim A. Majchrzak. Evaluating Cross-Platform Development

Approaches for Mobile Applications. Page 3. 2013. http://www3.nd.edu/~cpoellab/teaching/cse40814/crossplatform.pdf

/11/ The GTK+ Team. Features. Accessed on 10.4.2016. http://www.gtk.org/features.php

/12/ QtSharp project. Readme. 30.11.2015. https://github.com/ddobrev/QtSharp/blob/master/README.md

/13/ Riverbank. PyQt project Introduction. 2015. https://riverbankcomputing.com/software/pyQt/intro

/14/ Apache Cordova project. Documentation. Accessed: 17.4.2016http://cordova.apache.org/docs/en/latest/guide/overview/#plugins

/15/ Qt Documentation. Writing QML Extensions with C++. http://doc.Qt.io/Qt-5/Qtqml-tutorials-extending-qml-example.html

/16/ W3C. XMLHttpRequest Level 1, 30.1.2014. https://www.w3.org/TR/XMLHttpRequest/

/17/ The Qt Company. Qt 5.6 Documentation. QML Global Object. http://doc.Qt.io/Qt-5/Qtqml-javascript-qmlglobalobject.html#xmlhttprequest

/18/ The Qt Company. Qt 5.6 Documentation. Qt WebSockets. http://doc.Qt.io/Qt-5/Qtwebsockets-index.html

/19/ Apache Software Foundation. Javascript Tutorial. Accessed 10.4.2016. https://thrift.apache.org/tutorial/js

/20/ The Qt Company. The IDE, Qt Creator. https://www.Qt.io/ide/

/21/ The Qt Company. Qt 5.6 Documentation. Qt Android Extras, Notification example. http://doc.Qt.io/Qt-5/Qtandroidextras-notification-example.html

/22/ Raspberry Pi Foundation/ Raspberry Pi Tools. Git commit: 2b2d2046e6928da056207ad9b8a874209880d74d. https://github.com/raspberrypi/tools

/23/ Raspberry Pi Foundation, Raspbian Jessie Lite, Release date: 18.03.2016. https://www.raspberrypi.org/downloads/raspbian/

/24/ The Qt Company. Qt 5 super module. Tag: v5.7.0-beta1. http://code.Qt.io/cgit/Qt/Qt5.git/

SOURCE CODE

The source code of the prototype can be found under the Github repository:

https://github.com/WatchfulLikeACrane/SmartClient