Lappeenranta University of Technology
Faculty of Technology Management
Degree Program in Information Technology

Master's Thesis

Kimmo Kolehmainen

A COMMUNICATION MIDDLEWARE QUALITY ENHANCEMENT WITH QT FRAMEWORK

Examiners: Professor Jari Porras
M.Sc. Jussi Laakkonen

Supervisor: Assistant Petri Heinilä

ABSTRACT

Lappeenranta University of Technology
Faculty of Technology Management
Degree Program in Information Technology

Kimmo Kolehmainen

A Communication Middleware Quality Enhancement with Qt Framework

Master's Thesis

2010

104 pages, 28 figures, 1 table, 3 appendices

Examiners: Professor Jari Porras
M.Sc. Jussi Laakkonen

Keywords: PeerHood, Qt, Qt Mobility, middleware, quality, peer-to-peer, wireless

In this thesis, a Peer-to-Peer communication middleware for mobile environment is developed using the Qt framework and the Qt Mobility extension. The Peer-to-Peer middleware – called as PeerHood – is for service sharing in network neighborhood. In addition, the PeerHood enables service connectivity and device monitoring functionalities.

The concept of the PeerHood is already available in native C++ implementation on Linux platform using services from the platform. In this work, the PeerHood concept is remade to be based on use of the Qt framework. The objective of the new solution is to increase PeerHood quality with using functionalities from the Qt framework and the Qt

Mobility extension. Furthermore, by using the Qt framework, the PeerHood middleware can be implemented to be portable cross-platform middleware.

The quality of the new PeerHood implementation is evaluated with defined quality factors and compared with the existing PeerHood. Reliability, CPU usage, memory usage and static code analysis metrics are used in evaluation. The new PeerHood is shown to be more reliable and flexible that the existing one.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan koulutusohjelma

Kimmo Kolehmainen

Tietoliikennevälikerroksen laadun parantaminen Qt sovelluskehyksen avulla

Diplomityö

2010

104 sivua, 28 kuvaa, 1 taulukko, 3 liitettä

Työn tarkastajat: Professori Jari Porras
DI Jussi Laakkonen

Hakusanat: PeerHood, Qt, Qt Mobility, väliohjelmisto, laatu, peer-to-peer, langaton
Keywords: PeerHood, Qt, Qt Mobility, middleware, quality, peer-to-peer, wireless

Tässä työssä toteutetaan Peer-to-Peer tietoliikenneväliohjelmisto mobiiliympäristöön hyödyntäen Qt sovelluskehystä sekä Qt Mobility laajennusta. Peer-to-Peer väliohjelmisto – nimeltään PeerHood – on tarkoitettu palveluiden jakamiseen. Lisäksi PeerHood mahdollistaa palveluiden yhteyden muodostuksen, sekä toisten laitteiden monitoroinnin.

PeerHood konseptista on olemassa C++ toteutus Linux alustalle hyödyntäen sen tarjoamia palveluita. Tässä työssä PeerHood konsepti on uudelleen toteutettu pohjautumaan Qt sovelluskehyksen käyttöön. Toteutettavan ratkaisun on tarkoitus parantaa PeerHood väliohjelmiston laatua hyödyntäen Qt sovelluskehyksen ja Qt

Mobility:n tarjoamia palveluita. Kaiken lisäksi, hyödyntäen Qt sovelluskehystä, PeerHood väliohjelmisto voidaan toteuttaa helposti siirrettäväksi toisille laitteille sekä alustoille.

Uuden PeerHood:n toteutuksen laatua on arvioitu määriteltyjen laatutekijöiden avulla. Uutta sekä vanhaa PeerHood toteutusta on myös verrattu keskenään. Luotettavuutta, prosessorin käyttöä, muistin käyttöä sekä koodin laadun mittareita on käytetty arvioinnissa. Työssä toteutettu PeerHood vaikuttaisi olevan luotettavampi, sekä joustavampi kuin aikaisempi toteutus.

PREFACE

This project is done as a part of the PeerHood project in communication software laboratory of Lappeenranta University of Technology. I would like to thank for this great opportunity to be involved of this project and provided guidance for my thesis work. A special thanks to Petri Heinilä of good support and valuable information and opinions.

Without a good background support, accomplishing this work could have been much harder. Hence, I want to give my acknowledgements to my employer Digia Plc for support of studying and graduating. Especially, I want to thank my family; thank you my lovely spouse Eija and also mother and father. In addition, my friends have been supporting me a lot during this project.

Lappeenranta, November 2, 2010

Kimmo Kolehmainen

TABLE OF CONTENTS

ABBREVIATIONS

2D	2-Dimensional
3G	Third Generation
API	Application Programming Interface
CLR	Common Language Runtime
CPU	Central Processing Unit
FTP	File Transfer Protocol
FURPS	Functionality, Usability, Reliability, Performance, and Supportability
GCC	GNU Compiler Collection
GPL	General Public License
GPRS	General Packet Radio Service
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IDE	Integrated Development Environment
IMEI	International Mobility Equipment Identity
IMSI	International Mobile Subscriber Identity
IP	Internet Protocol
I/O	Input/Output
ISO	International Organization for Standardization
KDE	K Desktop Environment
LGPL	Lesser General Public License
LOC	Lines of Code
LSB	Least Significant Byte
MMS	Multimedia Messaging Service
moc	Meta-Object Compiler
MSB	Most Significant Byte
OS	Operating System
P2P	Peer-to-Peer
PH	PeerHood
PH1	PeerHood1, existing PeerHood implementation

PH2	PeerHood2, Qt based PeerHood implementation
QML	Qt Meta-object Language
RAM	Random Access Memory
RFC	Request For Comments
SDP	Service Discovery Protocol
SMS	Short Message Service
SSL	Secure Sockets Layer
STL	Standard Template Library
SVN	Subversion
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
WLAN	Wireless Local Area Network
XML	eXtensible Markup Language

1 INTRODUCTION

Developing a communication middleware can be very challenging. In many cases, there are needs for use a platform specific components and Application Programming Interfaces (API), like sockets, threads and processes. For decreasing platform dependencies in a software product, abstractions have to use much, which usually drives to more complex system design.

On desktop environment, there are several frameworks for helping to create applications without need of low-level platform APIs. The most well known frameworks are Java framework [1] and .NET framework [2]. By nature, these frameworks are virtual runtime environments, where applications are executed using byte code. On virtual runtime environments, applications are not compiled to a native code execution environment. Running application on runtime environment is not as efficient as execution of native application, because of byte code is compiled to the native CPU instructions at application runtime [2].

One interesting and potential cross-platform framework is a Qt (“cute”) framework [3]. At an early stage, the Qt was only cross-platform User Interface (UI) framework. However, it has been evolved over time to be much more than just a mere User interface framework. Current 4.6 Qt (14.6.2010) version offers components and tools for several different application layers, like platform independent concurrent network programming as well as Graphical User Interface (GUI) and much more.

Basic idea of the Qt framework is “Write once, compile everywhere”, which is very different approach than in Java runtime environment or Common Language Runtime (CLR), where application is executed on virtual machine. The Qt framework is written with C++ and it provides a set of interfaces and libraries for use of application developers. When using the Qt framework, application developers are not limited to use only Qt libraries. In addition, the Qt framework enables mixed use with platform specific libraries.

The 4.6 version of the Qt framework support numerous platforms where Qt applications can be targeted and compiled. For example, a same Qt application can be compiled into the Mac OS X, Linux, Solaris and Windows platforms. Excluding used platform specific implementations, if those are needed.

The Qt framework is not only available for the desktop environments. In fact, after Nokia bought Trolltech – the creator and owner of the Qt framework – it has use a lot of effort to bring the Qt framework available on mobile platforms as well. Nowadays, the Qt framework is also available on Windows CE, Symbian and Maemo mobile platforms [4]. In addition, there are several projects ongoing to extend the Qt framework with mobile device specific features. One of these projects is a Qt Mobility extension. The Qt Mobility extension provide interfaces to manage location information, contacts, connectivity and many others functionalities related domain of mobile devices [5].

One of the biggest problems of using the Qt framework has been very restricted license policy. After acquisition of Trolltech by Nokia, the license policy has changed to dual licensing. Currently, the Qt framework is available in GPL/LGPL [6, 7, 8] licensees and commercial license for proprietary use as well. This means that the Qt framework and sources of it are available for everyone's use under the common open source licenses. Anyhow, the closed licensing option provides ability to use and modify the Qt framework without publishing changes of it [6].

In this thesis, the Qt framework is used as a base technology in communication middleware programming. Rationale of this study is to enhance of existing Peer-to-Peer neighborhood communication middleware concept, which is called PeerHood [9]. PeerHood is a communication middleware, which is for sensing devices and services from wireless network neighborhood in proactive manner. The PeerHood provides transparent connectivity to service located in local or remote device.

In this thesis the term PeerHood is used to refer Peer-to-Peer neighborhood concept; PeerHood1 is used for the existing PeerHood implementation and PeerHood2 refers to a new PeerHood implementation based on the Qt framework. The new PeerHood

implementation – PeerHood2 – is based on the PeerHood1 and it is implemented during this thesis work.

1.1 Objectives

This work is part of a PeerHood project, which is research project of mobile Peer-to-Peer communication middleware [9]. Motivation of this work is to improve quality of an already implemented PeerHood middleware by utilizing the Qt framework. Quality enhancement is supposed to be done with reusing components and functionalities from the Qt framework and the Qt Mobility extension APIs. If needed, architecture of the PeerHood is modified and remade to be well structured with the Qt framework.

The main objectives to use sophisticated and mature framework is to get more defect free product with better extensibility and portability to other platforms. Current implementation of the PeerHood – PeerHood1 – is implemented with plain C++ targeted to Linux based platforms. The Qt framework and especially the Qt Mobility extension can enable a new use cases for the PeerHood. In addition, the Qt can enable language bindings for PeerHood API. Language bindings provide ability to use the PeerHood with other programming languages as well.

1.2 Scope and Delimitations

Even though the current PeerHood is implemented mainly on the Linux environment, this work does not focus on any specific platform. Idea is to use the Qt framework as enabler to create the PeerHood to be mainly implemented in a cross-platform manner. For prototyping and testing purpose, the desktop Linux environment is used.

Scope of this thesis is to do experimental research by implementing existing PeerHood1 implementation with utilizing the Qt framework and the Qt Mobility APIs. First objective is to evaluate how suitable the Qt framework is for communication middleware use. Second objective is to analyze implemented middleware – what kind advantages and disadvantages use of the Qt framework causes. The initial assumption is that the most of PeerHood functionalities can be implemented with the Qt framework to

be portable across different platforms without large modifications to it. In addition, the Qt framework is used to increase PeerHood quality and maturate by using Qt functionalities instead of implementing those by self.

2 SOFTWARE QUALITY

Quality is in the major role when talking about accomplishing software product improvements. Software quality must be defined and measured to follow up how good software is and prevent software quality regressions during software development process [10].

Software quality has been discussed a lot and it is a subject, which leads easily an almost endless debate of what that actually is. Even term quality is ambiguously defined in literature. A naive quality definition can be thought as a defect free product, thus if software product has plenty of functional defects, it does not fulfill it requirements anymore [10]. In the real world, software quality is a far away of defect free software. Actually, it is very likely that most of software have some known or unknown defects. Software quality is set of factors that alter in different applications in manner how these factors are prioritized. Quality factors can depend from software customer as well – things what they see important for the software product [10].

Developing a high quality software product is not always straightforward. Moreover, middleware modules, which many third party applications rely on, have to work how they are specified. In addition, developing middleware for mobile devices and embedded devices are even harder than on desktop environment. Usually, available resources are limited on mobile environment and application resource usage must pay attention. In addition, in mobile devices, available networks can vary a lot and device can easily run out of battery in intensive use.

There are many different things, which affect quality of a software product. Before software can be evaluated, the software quality must be defined. In this thesis is utilized a definition for software quality with different quality characteristics for giving a help to evaluate and measure software quality more accurately.

2.1 Quality Factors In Software

Concept of a software quality factors is not a new thing; in fact Boehm, Brown and Libow introduced quantitative evaluation of software quality in the 1976 [11] and after that McCall, Richards and Walters defined McCall's quality factors in the 1977 [12] and still these factors are valid and used to describe quality of a software product. McCall's quality factors are not only quality model that is presented. There are numerous different quality models like FURPS (Functionality, Usability, Reliability, Performance, and Supportability) developed by Hewlett-Packard. ISO standardization organization has also introduced ISO 9126 software quality factors [12].

It is very hard to select best quality model for use, in that many of these models are very similar and uses same characteristics. For this study McCall's quality factors seems to be feasible set of characteristics for examined quality and quality improvements in domain of this work.

McCall et al. identified three main aspects of a software product:

1. **Product Operation**
2. **Product Revision**
3. **Product Transition**

These aspects describe software behavior, flexibility for changes and adaptability to other platforms. Each product aspects are divided into several quality factories [12], which are show in Figure 1.

2.1.1 Product Operation

Product operation characteristics are related to software behavior – how well software behaves and follows the product specification without any abnormal operations [12]. These characteristics are the most important for the end-user point of view and these has biggest affects how the end-user experience used software.

- **Correctness** factor is about how well software operates as it is specified to do and how well software fulfills the requirements.

- **Reliability** factor is about maturity of software. How accurately software can be expected to operate specified tasks. In addition, reliability factor is about how long software can operate without any abnormal behavior.
- **Usability** factor is about learning curve of the software and effort required to operate it. In the middleware domain, this can also be how usable given APIs are and how well APIs are documented [13].
- **Integrity** factor is about secure control of software and used data. How well unauthorized data usage is protected.
- **Efficiency** factor is about software performance, amount of utilized resources and amount of code required to operate specified functions.

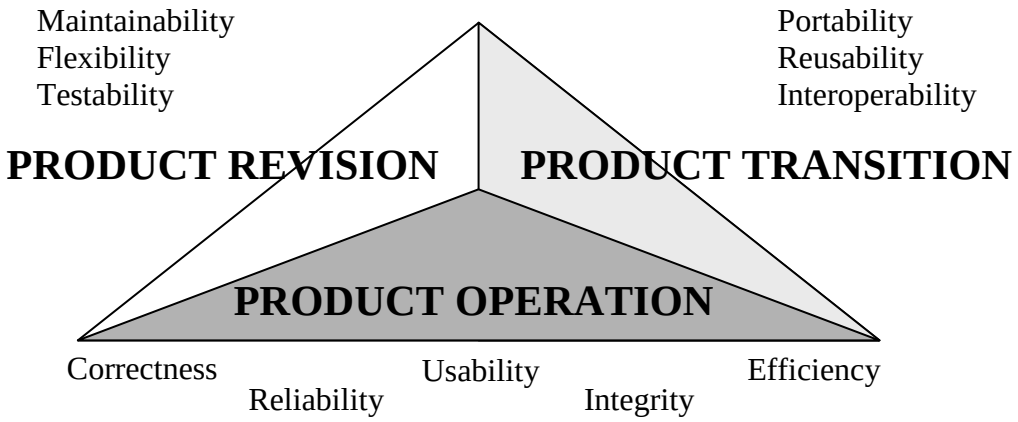


Figure 1. McCall's software quality factors, reproduced from [12]

2.1.2 Product Revision

Product revision characteristics define software flexibility of changes in software product. These characteristics are related to software architecture and they are very important to software internal quality and how easy changes can be done to the system [12]. The use of design patterns [14] in software design can have a big impact of these characteristics and make a system more adaptable and changeable.

- **Maintainability** stands roughly for effort required to identify and fix an error in the software. In addition, maintainability may include improvements and adaptation of the software systems to changes in the environment and in

requirements and functional specifications. A good documentation of the software and its structure can increase software system maintainability a lot [15].

- **Flexibility** is about how easy is make changes to software and how much it causes other changes in the software and its entire structure.
- **Testability** is about how well software can be tested and how much it requires effort for doing that.

2.1.3 Product Transition

Product transition aspect defines software adaptability to other environments. It is rather, that same software shall be used in different operating systems and different hardware's.

- **Portability** is characteristic which describe dependencies to underlying hardware and platform. Without a good abstraction, dependencies to underlying system can cause many changes to software when porting software to other systems. Even though the dependencies are abstracted well a platform specific parts need to be implemented in different platforms.
- **Reusability** is about component sharing between other applications related to software packaging and its functionalities. In addition, reusability can utilize internally with shared components
- **Interoperability** is about effort needed to couple system to another. In generally interoperability means ability of two or more systems or components to exchange and use information together.

2.2 Software Quality Metrics

Software metrics can be divided into three categories: **product metrics**, **process metrics** and **project metrics** [10]. This study concentrates only product metrics, which includes product characteristics like size of the software, performance and metrics for defined software quality factors.

In generally quality factors affected to software quality can be divided into two groups – directly measurable factors and only indirect measurable factors [12]. McCall's et al. defined metrics for the proposed quality factors. Many of these defined metrics are only subjectively measurable and cannot measure directly from the software [12].

For PeerHood quality evaluation, the McCall's quality factors are examined. Many of metrics for the McCall's quality factors proposed by McCall's et al. are very subjective and cannot measure directly [12]. In addition, many of these proposed metrics requires a long time analysis, which is not feasible in scope of this study.

Decision of rewrite the whole PeerHood with the Qt framework provides excellent opportunity to evaluate a new PeerHood implementation by comparing it with the old PeerHood implementation. With this approach, an influence of the Qt framework can be shown. For this reason, quality metrics are chosen so that they can be measured and compared together. Quality factors, which are more subjectively and cannot measure directly are discussed and analyzed as well. The following metrics are measured in both PeerHood implementations and results are compared together.

2.2.1 Static Code Analysis

With static code analysis, a several metrics for software quality evaluation can be measured [16]. The static code analysis can be used to measure software metrics like code complexity, size of code, comments on code and depth of code. In this case static code analysis refers quantitative measurement of implemented software, not code analyzers, which search common mistakes, like assign value instead of comparing values or use of uninitialized variables.

One of the most important static code analysis metrics is a code complexity. The code complexity has a big influence of code quality factors such as maintainability [15] and reliability [17]. In addition, code complexity has influence of code usability, in that complex code takes always more time to understand. Furthermore, the code size and amount of comments affect usability and maintainability of the software.

2.2.2 Efficiency

Software efficiency is related to software performance and system resource usage. Used resources can be CPU usage, Memory consumption and system resources, like network and file system usages. The CPU usage and memory consumption is used to as a metrics of execution efficiency of software [12]. In addition, memory consumption can be divided to heap, stack and virtual memory usages. The CPU and memory usages are important characteristic especially in embedded mobile devices, where available resources are very limited.

2.2.3 Correctness and Testability

Software correctness is quite subjective quality factor. It means that software behaves as expected without any abnormal behavior [12]. A real measuring of correctness would require continuous bug tracking and detailed requirements. In scope of this study, the software correctness is handled and measured of set of API tests, which are implemented during developing a new PeerHood implementation. With API tests and test coverage, testability can be increased [18] and software correctness can be proved partially. Test coverage is measured for tests, to get metrics for the testability [18].

3 PEERHOOD COMMUNICATION MIDDLEWARE

PeerHood is a communication middleware [19] for a peer-to-peer (P2P) [20] communication with a device neighborhood. The PeerHood concept is for mobile devices to monitor constantly services from other devices in the network neighborhood and provide transparent usage of services without any knowledge of the underlying network technology. It supports functionalities like:

- Detect other devices using different network technologies
- Discover services from other devices
- Advertise own services to other devices
- Monitor status of devices in network neighborhood

The PeerHood searches devices on network neighborhood in proactive manner by using available wireless network technologies. Ability to roam between different network technologies is provided by the PeerHood middleware as well. [9]

The PeerHood middleware is currently mostly targeted on a Maemo platform, which is based on the Linux operating system kernel and target to mobile devices such as Nokia Internet tablets and Nokia N900 mobile phones [21]. The PeerHood is also implemented on Symbian platform, but because of old limitations to create background server, it is not implemented as completely as on the Linux environment [22]. This thesis uses the Linux implementation as a reference PeerHood implementation and the Symbian implementation is not covered at all.

Uses of different network technologies are implemented with plugin implementation [9]. The PeerHood supports Bluetooth, WLAN and GPRS network technologies. Moreover, new plugins can be added if needed. In Figure 2 is shown basic concept of the PeerHood.

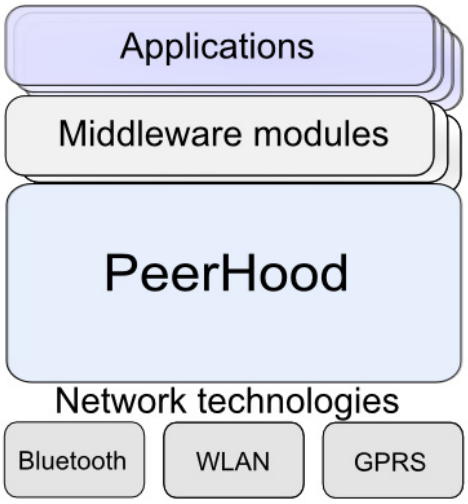


Figure 2. The concept of the PeerHood, reproduced from [9]

3.1 Key Requirements

Device discovery – system must be able to discovery other PeerHood capable devices within range and the same device neighborhood [22]. Device detection can be depend used network technology.

Service discovery – system must be able to discovery services from the local device and other PeerHood devices in the device neighborhood [22]. System must have capability to read service attributes as well with service discovery.

Service sharing – PeerHood must provide mechanism to register services and use them by applications or middleware components [22]. Services can locate on local or remote device. The PeerHood system must advertise registered services to other devices in a PeerHood neighborhood.

Connection establishment – PeerHood must provide ability of connect to one or more other PeerHood device in a PeerHood neighborhood [22]. Connect establishment must be transparent for used underlying network technology.

Active monitoring of a device – PeerHood must provide way to set a selected device in the PeerHood neighborhood under active monitoring. In the active monitoring state, a PeerHood client is notified when the device under monitoring is out of range or when it comes back in the range [22]. Proper response time and range are network technology dependent attributes.

Data transmission between devices – PeerHood must provide data transmission between connected PeerHood devices [22]. PeerHood should not take care of data being transferred. User of the PeerHood must take care of data endianess and word length of data.

Seamless connectivity – PeerHood should provide way to change used active network technology automatically if established connection weakens or breaks [22]. PeerHood should provide always the best possible connections for the user. Established connection should be possible to monitoring for detecting connection changes, which might cause change of used network technology [22].

Network management – PeerHood should be able to manage a specific network and events from the network [23]. In addition, PeerHood should check availability of network and get notifications of changes of the network.

Component management – PeerHood should provide events to PeerHood client of changes and suspensions of discovering functionalities [23]. The PeerHood operates on mobile devices where memory and power consumptions have to take care. Due to that, used device environment is dynamic. As, if network interface might go power saving state or it can be closed for freeing memory to other applications.

Communication concurrency base – PeerHood must support concurrent execution, in that multiple connections are used and they need to get execution time evenly [23]. The only exception for use of multiple simultaneous connections is if used network technology limits multiple connections on the hardware level.

Event interface – PeerHood must provide event interface for be able to notify dynamic changes to PeerHood client and itself [23].

Plugin architecture for networks – PeerHood must provide interface for its functionalities to plugins [23]. Network plugins implements abstractions of connectivity and device monitoring functionalities [23]. In addition, plugins handles device detection and service sharing.

User control – PeerHood could provide ability to control PeerHood functionalities [23]

- Is PeerHood active
- What services are provided
- What services are accepted

This is a new requirement and that is not yet implemented in the existing PeerHood implementation.

3.2 High-Level Architecture

In this chapter, a high-level architecture of the existing PeerHood implementation is explained. PeerHood implementation can be separated to three different components. These components are PeerHood library, daemon and network plugins. Network plugins actually contains several plugins for networking of different network technologies. PeerHood components are shown in Figure 3. and each component is explained more detailed in following chapters.

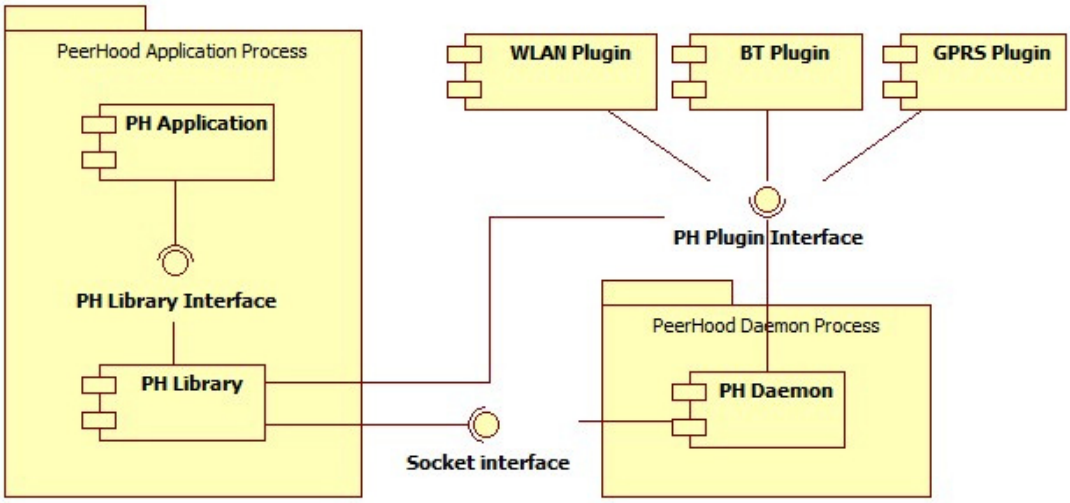


Figure 3. Main Components of the PeerHood, reproduced from [24] and current implementation

3.2.1 PeerHood Daemon

Daemon is the most important component in the PeerHood. Device and service discoveries and advertises local services to other devices in the PeerHood neighborhood are very heavy and resource consuming operations. The PeerHood daemon is used to decrease this heavy processing from an each application. In addition, when using background running daemon process approach, the information of remote services and available devices are already gathered when a PeerHood application is launched. That will reduce time required to get the PeerHood application ready for operating.

The PeerHood daemon gathers other PeerHood devices and their services from the network neighborhood. PeerHood capable devices and information are stored to neighborhood device registry. Registered device contains information about its services. The PeerHood daemon has another registry for local services, which daemon is advertise to other PeerHood devices. The daemon publishes a local socket interface for a PeerHood library.

With the socket interface, a library instance can request available devices and services from the daemon. In addition, new services can be inserted into daemon by the PeerHood library. The daemon publishes these registered services to other devices in

the PeerHood neighborhood. Services registered by the PeerHood library can be unregistered as well. The daemon utilizes network plugins for devices and service discovery from the PeerHood neighborhood.

3.2.2 PeerHood Library

The PeerHood library is a dynamic library component, which applications can include for use. The PeerHood library provides a PeerHood interface for use of third party applications and middleware components. Third party components can use the PeerHood only with the PeerHood interface. The PeerHood interface is available at Appendix 1.

The PeerHood library functionality is separated to be a client for the PeerHood daemon and providing ability to establish and manage connections between PeerHood devices. Like the PeerHood daemon, the PeerHood library uses network plugins. The PeerHood library uses network plugins indirectly through of abstractions. Used functionalities are a connection abstraction and device monitoring abstraction.

3.2.3 PeerHood Plugins

With PeerHood Plugins network specific implementations are done. With network plugins, support for new network technologies can be added easily. For each used network technology, the PeerHood has an own plugin. Usually one plugin create implementations for *MAbstractPinger*, *MAbstractMonitor*, *MAbstractConnection* and *MAbstractPlugin* interfaces.

3.2.4 PeerHood Applications

Applications can link against the PeerHood library and communicate with the PeerHood system through the MPeerHood interface (Appendix 1.). Applications can use the PeerHood system without any special knowledge of underlying network technology.

Common way to use the PeerHood from an application is to provide services to be available for other PeerHood applications in the PeerHood neighborhood or use services provided by other PeerHood applications. An application can use and provide services as well at the same time. Multiple PeerHood applications can operate on same device, hence used service does not always locate on a remote PeerHood device. In addition, with the PeerHood, device can be selected to be continuously monitored. If monitored device moves out of the range, the application is notified of it.

4 QT FRAMEWORK

The Qt framework is a cross-platform application development framework. Cross-platform support for the Qt framework is done a way of "Write once, compile everywhere" principle, which means that same source code, can be used on several platforms. With this principle, application is executed on native environment without any virtual execution environment. The current 4.6 Qt framework is available on multiple desktop and mobile platforms. These are following platforms [4]:

- Embedded Linux
- Mac OS X
- Windows
- Linux/X11
- Windows CE/Mobile
- Symbian
- Maemo

The Qt framework provides unique APIs, which are used as platform abstractions. The platform specific implementations are wrapped behind of these APIs [25]. Because of used abstractions, usually application developer does not need to care of target platform. Of course, there are some exceptions and some restrictions as well. For example, a desktop application can be very hard to get working on mobile platform without any changes of it. At least some changes to application UI might need to be done to keep application user experience in good level.

Maybe the most well know use of the Qt framework is in a KDE Project, they have long history with the Qt framework and Trolltech Company. The KDE project has used the Qt framework since 1996 when KDE project was started [26, 27]. The KDE project and Trolltech made an agreement, where Trolltech promised to keep the Qt framework as a free for the KDE project [28]. That agreement was for the KDE project, so they were able to rely on that the Qt framework will be free software in use of the KDE Project. Later on, the Qt framework published under GPL license [6], which made that agreement useless and available the Qt framework free to use for everyone.

The Qt framework is usually known as a cross-platform graphical user interface (GUI) framework. That has been true a long time ago. Nowadays, the Qt framework is much more than just a sophisticate graphical user interface framework. The Qt framework provides platform independent interfaces for many other purposes than GUI programming use as well [3]. The Qt framework is component based; these main components are presented in Figure 4. Multithreading and 2D Graphics Canvas are not separate components, instead they are wanted to emphasize in the figure and they belongs groups of Core and GUI modules. In subchapters, Qt components related in scope of the PeerHood middleware are presented and basic Qt principles and models are described. Additionally, an interesting Qt extension [5] – the Qt Mobility – is introduced and some APIs of the Qt Mobility extension are described more detailed.

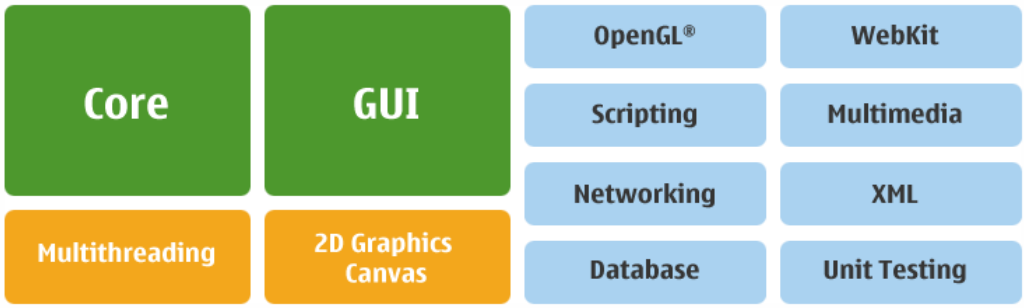


Figure 4. Qt Component Overview [3]

Even though the Qt framework is written with C++ language, it does not limit all applications to be written with C++ language. There are available numerous language bindings for the Qt framework. Trolltech official support Java and JavaScript language bindings and bindings for Python, PHP, Ruby and .NET are provided by third parties [28]. With these various language binding options, the Qt framework is even more portable and versatile.

4.1 Qt Object Model

The Qt framework includes a built in object model, which is heart of the Qt framework. The most important class for this Qt object model is a QObject class. The QObject has several roles of the Qt object model and the QObject is base class for almost all objects inside of the Qt framework [29].

Inside of the Qt object model, QObjects are related together in a Parent-Child relationship. Internally, QObjects organizes relationship within an object tree [29]. The object tree is a key enabler of an intelligent object management inside of the Qt framework. With the object tree, the Qt object model can provide type of semi-automatic memory management by enabling automatic child object deletion when deleting the parent object. The parent for the QObject has to define explicitly and habitually parent is given as the parameter of an object constructor. QObjects can query all child objects that belong to it and parent object for itself. Note that the parent-child relationship is not a same as inheritance in object-oriented languages. It is just connection between two QObjects.

The Qt object model provides more flexibility and better object runtime information of objects than standard C++. With the Qt, properties of an object are query able on application runtime [29]. Some compilers make available object properties functionality as well. However, for getting object properties working a way of cross-platform it is done by using a Qt's Meta-Object System (moc) [30] in the Qt framework.

Advantage for use of meta-object system is to keep dependencies for any compiler specific options as minimal as possible [30]. Moreover, this system enables dynamic runtime property declaration to Qt objects. The Qt meta-object system is based on three things [30]:

- The QObject, which provides base class for all classes that, can take advantage of meta-object system.
- Q_OBJECT macro, which must be defined in private scope at the beginning of a class definition. The Q_OBJECT macro is used to enable meta-object features, such dynamic properties and signals and slots mechanism.
- The Meta-object compiler, which generates meta-object code for each class, which declare the Q_OBJECT macro. More about compiling and the meta-object compiler is discussed later on.

Although, the Q_OBJECT macro is used to enable dynamic properties and signals and slots, the Qt documentation recommends use that macro for all QObject classes,

regardless of is features provided by the meta-object system used. E.g. the Qt provide qobject_cast method for casting QObjects, however outcome of that method is undefined if Q_OBJECT macro is not used in object to be casted.

Every QObject instance has a unique identity in the Qt object model. Use of unique identity causes some limitation for classes inherited from the QObject. In that, copy constructor and assignment operator has been disabled by implementing them in private scope by default. Therefore, all objects inherited from the QObject have to move between methods using pointers or object references.

The Qt framework has two event systems, which provides ability to deliver events between objects. The Qt object model enables these event mechanisms. These event models are a Qt event system, which allows sending and posting events to registered event listeners and a *Signals* and *Slots* mechanism. Signals and Slots mechanisms are used to communicating between QObjects. These both models are introduced more detailed in following chapters.

The Listing 1 shows a basic example of the Qt style class definition, which is inherited from the QObject and it utilize signals and slots mechanism. Important parts are numerated inside of comments in the example code. All these numerated lines are explained.

Listing 1. Example class inherited from the QObject

```
#ifndef MYEXAMPLE_H
#define MYEXAMPLE_H

#include <QtCore/QObject>       // 1.
#include <QtCore/QDebug>

class MyExample : public QObject
{
    Q_OBJECT                    // 2.
public:
    MyExample(QObject* parent=0)// 3.
        : QObject(parent)
    {}

public slots:                   // 4.
    void receiveSignal()
    {
        qDebug("signal received");
    }

signals:                        // 5.
    void mySignal();            // 6.
};

#endif // MYEXAMPLE_H
```

1. Include of the QObject. Defined MyExample class is inherited from the QObject, which is located inside of the QtCore module.
2. To enable use of signals and slots mechanism, the Q_OBJECT macro must be defined in the private scope at the beginning of the class definition.
3. The QObject as the parameter of the MyExample class constructor is forwarded to the QObject constructor. Delivering the parent object to the QObject constructor enables instance of the MyExample class to be part of parent object tree.
4. Public slots scope for defining slots in the MyExample class
5. Signals scope for defining signals, what MyExample class can emit
6. A definition of a MySignal. The signal does not contain any parameters.

4.2 Signals and Slots

One of the interesting tools in the Qt framework is Signals and Slots mechanism [31]. It is powerful seamless connection system between QObjects and subclasses of the QObject. With the signals and slots mechanism, events between objects can be sent without any known of receiver object or objects. The Signals and Slots mechanism is one kind replacement of callback mechanism by providing type safe notifications to event receivers called as slots. By nature signal-slot connection is many to many

connections, thus multiple slots can be connected to same signal. Furthermore, multiple signals can be connected to a same slot as well. Figure 5 shows how signals and slots can be connected. In addition, signals can be connected to other signals. When connecting signals to slots the function signatures must match together. However, exception is a case where signals have more parameters than a slots, then slots is called and extra parameters are ignored. The signal and slots are loosely coupled and thus, connections to unavailable or misused signal and slots does not cause any compile time errors. The Qt system prints out warning message at the runtime if connection is failed.

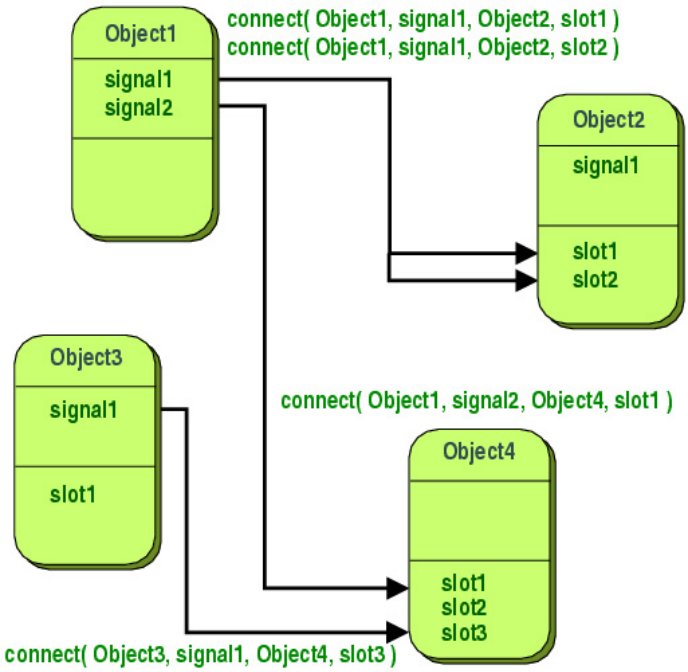


Figure 5. Signals and slots model [31]

Classes inherited from the QObject can define signals and slots. When defining slots or signals in a class definition, the meta-object system must be enabled with the Q_OBJECT macro at the beginning of the class definition [31].

For the Signals and Slots mechanism, the Qt has extended C++ keywords with extra words. New keywords are *signals*, *emit* and *slots*. Keywords signals and slots are used in a class definition like visibility scope operations. The slots keyword is used with public, protected or private visibility scope operators. For the signals this visibility scope operators are not used. Section 4 and 5 in the Listing 1 shows how slots and

signals keywords can be used. emit keyword is used in code when defined signal is wanted to emit.

Signals are only abstract methods without any implementation. The Signal definition is used as a template what kind function signature has to have for a receiver slot. All signals, which class can emit should be defined in the scope of signals keyword. Nevertheless, inherited signals can be used without redefine. In the Listing 1, the section 6 is shown how to signals can be defined. In the example, *mySignal* signal is defined without any parameters. Signals can never have a return value hence void return type must be used.

Slots can be thought as normal methods, which can be connected with signals. Only exception to normal methods is how they are introduced in a class definition. Slots must be defined in scope of slots keyword with a visibility keyword. Defined slots can connect to signals and when connected signal is emitted slot is executed. However, slots can be called also like normal method depending scope of visibility. Slots can be invoked from any component regardless of scope of visibility via signal-slot connection. With signal-slot connection visibility of a slot defines only access level of connection establishment to slot. Thus, slot in private scope can be connected only inside of the class. When slot is called from signal emit, the return value of slot is ignored. Usually, return values of all slots are defined to be void. Slots can be defined to be virtual as well.

When using the signals and slots mechanism, a developer cannot ever assume that slot is called directly after signal is emitted [28]. In that, signals can be connected to slots using direct connection or queued connection. Furthermore, when signal is emitted from different thread than receiver object is the slot connection is always made in queued connection.

4.3 Event Loop and Events

Almost all Qt applications are event-driven applications, excluding the simplest hello world console application. Events for an application are delivered from the Qt's event

loop [32]. By nature, the Qt event loop is like Reactor design pattern [33], which is used to provide events to registered event handlers.

Event loop can be controlled with a QAbstractEventDispatcher. Global instance of the QAbstractEventDispatcher can be reached from *QAbstractEventDispatcher::instance()* method. The Qt allow implement an own event dispatcher as well by inherit the QAbstractEventDispatcher class. For using own event dispatcher, it must be created before Q(Core)Application is created.

The main event loop is executed in the applications main thread and it is responsible to keep the application running until application exit. The main event loop must run in the main thread in the Qt. At the beginning of Qt application startup, the main event loop is usually started. A QCoreApplication and a QApplication inherited from the QCoreApplication are used to start main event loop. Both have a method *exec()*, which is generally synchronous method all over Qt classes.

Usually main function of a Qt application is implemented so that it does first some initialization. After application is initialized an *exec()* method is called from the QCoreApplication or the QApplication object. The *exec()* method starts the Qt main event loop. Habitually, the application exits directly when the main event loop is quit and the *exec()* method call returns back to main function.

The QApplication is part of a QtGui module and it is used when an application uses Graphical User Interface [28]. The QCoreApplication is defined in the QtCore module and it is used in case when application is console application, like background daemon.

Because of Qt event system is based on event loop, the Qt cannot deliver events before the main event loop is started [32]. This consist all events and signals based on queued connection. However, direct signal connection is possible because those are executed without event loop. Furthermore, Qt dialogs can be used before event loop is started, in that they uses own event loop to receive events.

Events in the Qt system can be delivered to its receiver in two ways [32]. These options are post events and send events. The differences between these two methods are that posted events uses main event loop and sent events are delivered directly without event loop to receiver. In addition, posted events must be allocated from the stack and the ownership of the event object is taken immediately when post is done.

4.4 Qt Core Module

The Qt core module named as QtCore is module, which provides core functionalities to other Qt modules and Qt based applications [25]. All fundamental Qt classes, like QObject, are included in the core module. The Qt core does not include any classes related to graphical user interfaces. In following subchapters, tools and functionalities provided by the Qt core module are introduced.

4.4.1 Data Types, Containers and Smart Pointers

Primitives

The Qt core module contains a several extended and improved data types from C++ types [25]. As all other Qt classes, provided data types are implemented in cross-platform manner. One pitfall of cross-platform development is sizes of primitive types. Usually, the native integer size depends on system bit wide. For example, int primitive can be 16 bit in some systems and 32 in some other system. For this reason, the Qt framework provides size defined primitive types. The size defined primitives are guaranteed to be same size on all platforms [25].

Strings

Maybe the most used non-primitive data type is string. A QString is Qt implementation of string type. It has a full Unicode [34] support on all platforms where the Qt is supported. Unicode 4 is used as a native character set for the Qt framework. Moreover, the Qt framework Unicode support provides detection of Windows so the Unicode support is available even though Windows platform does not support it natively. The QString implementation uses an implicit sharing [25] like many other Qt classes. The implicit sharing optimizes memory usage of strings.

Pointers

The Qt Core module provides also useful helper classes for safer pointer use [25]. Everyone who has developed software with C or C++ knows how error prone using pointer can be. For making things much easier, the Qt framework provides a set of smart pointer classes. These classes help avoiding memory leaks of dynamically allocated objects and protect against of dangling pointers.

QPointer class provide guarded pointer for classes based on the QObject. The QPointer behaves like a normal C++ pointer, except it is automatically set to 0 when referenced object is deleted. Hence, the QPointer is useful pointer when need to refer some pointer, which is owned by someone else. With the QPointer, the use of invalid pointer cannot happen. The QPointer can be used only with subclasses of QObject.

A QScopedPointer is a bit different kind pointer class than the QPointer. Use of the QScopedPointer is not limited just QObject based classes. The QScopedPointer simplify use of heap objects in a particular scope. Conventionally the QScopedPointer itself is a heap object in the scope of a method. When execution of the method goes out of the scope, the QScopedPointer instance get deleted and so is deleted allocated memory that the QScopedPointer instance is referring. Unlike the QPointer, the QScopedPointer refers a memory, which must be owned. In addition, the QScopedPointer can contain any kind of objects, not just the QObject based objects. One exception is traditionally allocated arrays, which must be stored to QScopedArrayPointer. The QScopedArrayPointer uses *delete[]* instead of the delete operator for deleting contained memory.

Containers

The Qt framework provides a set of generic template based container classes [35]. C++ Standard Template Library (STL) provides same kind set of containers, but the Qt versions are designed to be easier and safer to use. The Qt container set contains optimized sequential and associated containers. All Qt container classes use implicit sharing to decreasing memory consumption [35]. In addition, containers are reentrant and in situation where container is read-only, it is thread-safe as well.

Qt containers can be traversed either Java-style iterators or STL-style iterators [35]. The STL-style iterators are a bit efficient than Java-style iterators are, although the Java-style iterators provide high-level functionality and they are easier to use. With the STL-style iterators, a STL's generic algorithms can be used [35]. A QtAlgorithms header offers access to STL's generic algorithms [25].

4.4.2 I/O Devices, Data Array and Streams

QIODevice

A good data handling classes can provide a great help for the communication middleware use. The Qt framework consist nice abstraction of I/O device, which is base of socket handling as well. A QIODevice is a base class for all I/O devices in the Qt framework [25]. The QIODevice provides common interface for devices that support reading and writing blocks of data. It is abstract class and cannot be instantiated directly. The QIODevice based classes are made for handling input and output to and from external devices, files or processes. In addition, instead of handling some regular I/O device the QIODevice interface can be used to write and read data in QByteArrays as well.

The QIODevice provides support for two types of devices – random-access devices and sequential devices [25]. Random-access devices support seeking to arbitrary position using *seek()* method. Opposed to random-access devices, seeking are not supported with sequential-devices, hence data must be read in one pass. Type of device can be determined from QIODevice with a method *isSequential()*.

Several device types are always asynchronous by nature. Asynchronous write and read operations are returned immediately and operation complete later on. The QIODevice contains signals *readyRead* and *bytesWritten* to react when a new data is available for read or data payload is written to device. Methods *bytesAvailable* and *bytesToWrite* are usually use with these signals to find out amount of data available or sent to device.

The QIODevice allow to make asynchronous write and read operations to be synchronous. This can be done with *waitForReadyRead* and *waitForBytesWritten*. With

these methods, a calling thread is blocked without going back to event loop, which makes possibility to use QIODevice without an event loop or in separate thread. Subclasses of QIODevice can provide other device specific blocking methods as well.

QByteArray

Since the QString is based on 16-bit Unicode characters, it is good to have some type, which can contain 8-bit data. For that reason, a QByteArray can be used [25]. The QByteArray is data storage for both raw bytes and traditional 8-bit '\0' terminated strings. The QByteArray is more advanced data container than C++ arrays are and the QByteArray always ensure that the '\0' terminator follows the data. In addition, like many others Qt containers the QByteArray use implicit sharing to reduce memory usage and avoid unnecessary memory copies.

When using the QByteArray it is good to remember that when using raw bytes the '\0' termination is added [25]. Without knowing that issue, it can cause unexpected behavior in some cases. The QByteArray can use like C++ arrays by pointing specific array position with [ ] operator. Like C++ arrays, the QByteArray uses 0-based indices as well.

QTextStream and QDataStream

For powerful I/O device usage the Qt framework offers two stream classes for use to application developers. With streams, the Qt framework handles the most platform differences such as endianess and use of proper line endings [25]. Therefore, application developers do not need to take care as much of platform compatibility. QTextStream is for reading and writing a text with I/O device. Moreover, for reading and writing binary data with I/O device a QDataStream can be used. With both stream operators << and >> can be used to read and write data in streams. Streams provide data serialization of C++ primitive types and basic Qt types. Serialization of more complex classes depends on class implementation. Many Qt classes' offers overridden << and >> operators for stream them into streams.

By default, the QDataStream handles data internally in most significant byte (MSB) order format [25], which gives better interoperability with other devices. Using MSB format ensures the cross-platform compatibility. Binary stream of encoded information is 100% platform independent. Nevertheless, data encoding can be changed to little ending format, which is not recommended as it breaks the platform compatibility.

4.4.3 Concurrent Programming

Processes

Processes and threads provide common approach to implement application concurrency. The Qt framework provides convenient and cross-platform classes to handle processes and threads. In addition, the Qt framework comes with the advanced threading options, like thread pool functionally [33]. Concurrent programming has been challenging with native C++ environment, in that using processes and thread are handled in a different ways by different operating systems. Hence, porting application from environment to another has been challenging.

A QProcess is class for controlling other processes. It provides interface for spawn processes and communicate with them. Even though, the QProcess is cross-platform implementation for handling processes, some actions depends user permissions. For example, process kill and terminate requires PowerMgmt capabilities on Symbian platforms. If required capability is missing, action will fail [25].

Threads

Using threads is a second – a much lightweight – solutions to implement concurrency in application. Threads can be created and controlled in the Qt framework through of a QThread class [25]. Qt thread model permits the prioritized of threads as well. When using threads in Qt, the signals and slots mechanism and event posting can be used. It is remarkable, that when using signals and slots between different threads with automatic connection type, the queued connection type is always used. The QThread has a virtual *run()* method which is executed after thread is started. The *run()* method is a thread entry point, like *main()* function is for the application. The QThread emits signal started

when execution of thread is started and finished signal when execution is finished. Thread execution ends when *run()* method returns. Default implementation of the *run()* method calls only *exec()* to start event loop for the current thread.

When using threads in the Qt, there are some limitations, which must be followed [25]. Each threads consist own surroundings, which means that each thread can have own event loop and each created QObject based objects belongs to current thread by default. QObject can be moved to another thread with *moveToThread()* method. First limitation is, that child objects cannot be in a different thread than the parent. Second limitation is that, all event driven objects may only be used in one thread. Meaning that, for example, socket cannot connect or timer cannot start from another thread where object is. Third limitation or more like requirement is that all objects created in a thread must be deleted before QThread instance is deleted. In addition, it is good to remember that if event loop is not running on a thread, objects cannot receive posted events or signals.

Thread Pool

Third option to implement concurrency using the Qt is use a thread pool based solution [25], which recycles threads. When using thread pool, there are no needs to create a new thread every time. Creating thread is known as operation, which is non-deterministic and time-consuming operation [33]. Using thread pool with the Qt is very straightforward. First subclass of a QRunnable must be implemented with implementation of virtual *run()* method offered by the QRunnable. After that, runnable class can be started with global thread pool instance. Each Qt application has one global QThreadPool instance. The QThreadPool provide interface for configuring thread pool options. The QRunnable provide control of class auto deletion. If QRunnable auto deletion is set on (the default), then the runnable object is deleted automatically after execution is accomplished.

Mutexes and Semaphores

When discussing about concurrent programming it is always important to remember of data synchronization and protection. Mutexes and semaphores are usually used for data protection and synchronization between concurrent instances. The Qt comes with mutex

and semaphore implementations [25] and a few nice utility classes. Utility classes give ability to use mutexes more safely and more optimized. QMutex represent Qt mutex implementation and a QSemaphore offers semaphore functionalities.

A QMutexLocker is one of mutex utility classes. The QMutexLocker implements common Scoped locking design pattern [33]. The Scoped locking design pattern is intent to simplify mutex locking and especially unlocking. The QMutexLocker is a simple class, which is meant to be allocated from heap. When heap object goes out of scope, the mutex is released by the QMutexLocker object destructor. With scoped locking solution, mutex is always released when method returns.

Second utility class is QReadWriteLocker, which can be used to increase performance of multithreading system. The QReadWriteLocker utilize Read-Write Locking design pattern [33]. The Read-Write locking is intent to increase availability of data. When using some common data from multiple threads it must be protected from changes. The idea of Read-Write locking is that multiple instances can lock data for reading at the same time. Only if data is wanted to change, the write lock is acquired and then other instances will not get access to it until lock is released again.

4.4.4 Timing

Using the Qt, a timing functionality can be implemented with two ways [25]. The QObject itself provides methods for start object timing. In this way, a subclass of the QObject needs to implement virtual *timerEvent()* method for receiving timing events. Timing events are wrapped inside of a QTimerEvent class.

Second option to create timed operation is use a QTimer class. If timed operation need to be done only once, the QTimer provides a static *singleShot()* method for simplifying use of timing. When continuous timing is needed the QTimer must be instantiated and connect *timeout* signal to some slot. The *timeout* signal is emitted when timer interval is elapsed. The QTimer provides basic interface for controlling timing and settings of timing.

Accuracy of timers depends on the underlying operating system and hardware. Most systems support at least one millisecond resolution. It is guaranteed by the Qt, that timer never timeout until exact specified timeout value is reached.

4.4.5 Plugins

There are two different APIs for creating plugins with Qt. High-level API is for creating extension plugins for the Qt itself [25]. Moreover, Low-level API is for creating application extension plugins. Because of the scope of this study, the High-level API is not handled at all.

The low-level plugin API is more interesting in application developer point of view. API enables run-time loading of shared library plugins into application. For implementing use of dynamic extensions into application, the application developer must create a pure virtual interface and declare it with Q_DECLARE_INTERFACE macro. The defined macro publishes interface to meta-object system. After interface is defined, a QPluginLoader can be used to load shared libraries from wanted location. Finally, with qobject_cast method, the loaded plugin instances can validate to be correct type.

Finally, implemented plugin object must be exported with Q_EXPORT_PLUGIN2 macro. After these steps, the application can load compiled dynamic library from location where binaries are deployed into. The implemented plugin class must be derived from QObject, if implemented pure virtual class is not. When plugin is loaded into application, the QPluginLoader verify that plugins are build against same Qt version than the application has.

4.4.6 Settings

Several applications need to store or retrieve some configuration information for application. There is not available common way to handle application settings in different platforms. The Qt provides an abstraction for handling settings in same way in all different platforms [25]. QSettings class enables store and read settings without

taking care of underlying platform. The QSettings is persistent map of key/value pairs. It is reentrant and same settings can be read or write from multiple thread or processes at a same time. Changes performed to settings are not visible for other processes until a *sync()* method is called for settings. The *sync()* method is automatically called in settings destructor. In Addition, *sync()* is called in regular interval by the event loop. Hence, usually application developers do not need to take care of that. In a same process, changes made to settings will be immediately visible to other settings objects, which are using the same settings.

4.5 Qt Network Module

A Qt network module provides tools for programming portable network applications. The Qt network module is named as a QtNetwork and it contains low-level and high-level networking tools [25]. High-level networking tools provide APIs to handle HTTP (Hypertext Transmission Protocol) and FTP (File Transfer Protocol) client connections without using any low-level networking. In scope of this work only low-level networking tools are included.

Using networking in communication middleware has been a bit challenging, since network sockets are generally very low level APIs and very platform depended [33]. The Qt provides fully cross-platform network sockets for local host connections, TCP (Transmission Control Protocol) connections, UDP (User Datagram Protocol) connections and SSL-connections (Secure Sockets Layer). In addition, it provides classes for handling incoming TCP connection requests and incoming local socket connections. Additionally, the Qt offers a socket abstraction of common socket functionalities for all socket types.

All Qt sockets are inherited from QIODevice, hence stream classes makes socket use very convenient and straightforward. When using QDataStream, used byte order is already managed by the Qt to be in platform compatible MSB (Most Significant Byte) format [25].

Sockets in the Qt are always asynchronous it however, there are methods for blocking execution without going to event loop. Signals and slots mechanism gives a nice ability to handle asynchronous socket operations. With the signals and slots dependencies are loosely coupled, which gives better flexibility of change.

4.6 D-Bus Module

D-Bus is a message bus system, which enables inter-process communication between applications [36]. The D-Bus is mostly used in different Linux and UNIX operating systems. A QtDBus module provides ability to use D-Bus functionalities in these platforms. The QtDBus component is one of the platform specific Qt components.

Applications using the QtDBus module can share provided services to remote applications by exporting objects [25]. Moreover, shared services exported by other applications can be used. The QtDBus module extend the Signals and Slots mechanism by providing ability to connect signals from remote application, as well as connect local signals to remote slots.

In software portability manner, use of the QtDBus is not recommended. Certainly, it could be used to wrap some platform specific functionalities to Qt based application. In fact, using D-Bus is required in several cases. For instance, handling Bluetooth with BlueZ on Linux is relying on use of the D-Bus message system.

4.7 Qt Test Module

The Qt framework includes unit-test framework for the Qt based applications [37]. With a QtTest module, application developers can write different kind test sets for applications. The application developer can write basic unit tests, data-driven unit tests or GUI tests for applications or interfaces. Furthermore, the QtTest library provides tool for monitoring emitted signals for testing purposes.

In Addition, benchmark tests with varying data can be implemented with the QtTest library. The benchmark tool provides way to measure execution time for the specific

operations. The results of the benchmarking can be plotted to graphical form with *qtestlib-tools* [37].

4.8 Development Framework Impact

A Qt application development environment can be almost same than C++ application development environment. Addition of the Qt libraries, two tools for compiling a Qt application is needed. First tool is a qmake tool, which is used to generate platform specific makefiles from Qt's project files (.pro and .pri files). Second tool which is required for compile Qt application is Meta-Object Compiler (moc) tool [30].

Other additional development tools are not needed, however, in Qt application development, a Qt Creator IDE is a good tool intended to Qt cross-platform development. Qt Creator Integrated Development Environment (IDE) comes with a cross compiling environment, which helps application development to different environment, especially mobile environments.

4.8.1 qmake

A qmake is a tool, which simplify build process on different platforms. With the qmake, the Qt can offer cross-platform ideology in project configuration and build system as well [38]. In that, Qt project files and qmake unify varying make file systems on different platforms. The qmake generate natively used make files from project files. For example, on Linux environment, the qmake generate Makefiles and on Symbian environment, it generates bld.inf files and pkg-files for creating deployment packages. The qmake can be used for any C++ software projects whether is it a Qt based project or not [38].

The qmake uses project files, which defines project to be build. The project files use .pro file extension. Moreover, .pri file extension is used for files, which can be included project files. The qmake provide sophisticated way of define software projects in multi-platform usage. The qmake includes a set of operational functions, variables, and conditional statements [38].

4.8.2 Meta-Object Compiler

Meta-Object compiler (moc) is for preprocessing extended C++ code from used Qt code in the project [30]. The moc tool handles all header files from the project and generates meta-object code from classes, which declares the Q_OBJECT macro. The meta-object code is required for the signals and slots mechanism, the object run-time information and the dynamic property system.

Meta-Object system is very transparent for application developer. Moreover, usually application developers do not need to take care of meta-object system. Except, declare of the Q_OBJECT macro in class definition. Generated C++ files must compile and link with class implementations, nevertheless qmake system can automatically include meta-object codes into build process.

4.8.3 Compiling

Compiling Qt application depends on target platform. A make process can vary between different platforms. Usually, the GCC (GNU Compiler Collection) compiler can be used on each platform. However, used compiler is not limited anyhow. Hence, Qt applications can be compiled with third party compilers and vendor-supplied compilers as well. More about supported compilers and platforms can be found from the Qt's supported platforms web page [4].

4.9 Qt Mobility Extension

The Qt Mobility Extension is targeted to being a key enabler for use of the Qt framework effectively in cross-platform mobile application development [5]. The Qt Mobility is a collection of APIs. The Qt Mobility version 1.0 was released on 27.4.2010 and last update 1.0.2 was released on 27.7.2010. All APIs of the Qt Mobility extension are available on Symbian and Maemo platforms [39]. In addition, with a few missing functionalities the Qt Mobility Extension is available for Windows CE/Mobile, Linux, Windows and Mac OS X desktop environments [40].

The Qt Mobility API provides a large scale of functionalities mostly targeted on mobile application development. There are available functions like handle sensor information of device, control contacts information, retrieve location and using SMS, MMS or email functionalities [39]. In addition, there are APIs for Bearer Management, System Information and Service Framework, which are discussed more detailed in following subchapters.

4.9.1 Bearer Management

Purpose of the Bearer Management API is for control connectivity state of device [39]. Using this API, application developer can access information of what kind bearer types there are available or are device currently online. Moreover, with the Bearer Management, network interfaces can be started or stopped. However, network configurations itself cannot be managed with this API, as configurations can only be used.

The Bearer Management API is the first Qt Mobility Extension API, which is already migrated into main Qt Network library. The Bearer management API is included in 4.7 Qt version. The release candidate of Qt 4.7 is already released on 12.9.2010 and probably the final release will be released soon.

Network configuration contains information of network interface and configuration for that network interface. Network configuration information is used to specify network more detailed, like how network interface can be started [39]. For example, in WLAN connection, access point details, such a data encryption and credential information for establish a connection is needed. The Bearer Management API provides information of available network configurations and control of start and stop specified network configuration for communication. In addition, the Bearer Management API enables actively monitoring changes of network configurations or device connectivity status.

4.9.2 Service Framework

The Qt Service Framework is a concept, which enables loosely coupled service usage. Services are independent components (plugins), which allow clients to perform specified operations [39]. Service can be registered to the service framework, where other application can discover needed services by name and version of the service. If service is found, application can load and use defined operations of the service. Services can be added or removed at runtime to service framework. Services are installed via XML file, which contains Meta data of service and location where service can be found. The Meta data contains available interfaces, descriptions for the interface and capabilities for the interface.

4.9.3 System Information

A system information API enables a common way to retrieve information and capabilities of underlying system and hardware [39]. The System Information API consist a several categories, which information are provided. These categories are device information, display information, network information, screen saver information and storage information. In addition, there is also general information category. In scope of this work, device information and network information are the most interesting options.

Device information is available through of a QSystemDeviceInfo class. That class makes available information of underlying device. With the QSystemDeviceInfo class user can retrieve information of device IMEI (International Mobility Equipment Identity) code and IMSI (International Mobile Subscriber Identity) code, which can be used as unique identifiers. Also this API provides information of currently used operation profile (e.g. silent profile, loud profile or normal profile) and power state status. The power state status indicates is device operated for example in battery mode or fixed power mode. Furthermore, battery level can be retrieved. Finally, the QSystemDeviceInfo can offer asynchronous notifications of changes in battery, profile or power states.

Information of general mobile network can obtain with a QSystemNetworkInfo, which belongs to the network information category. The QSystemNetworkInfo class provides interface, which can be used to retrieve information of network name, network cell id, current location code and home network and country codes. With the QSystemNetworkInfo class, network signal strength can be read and monitored in asynchronous way. Moreover, the network status can be retrieved with the network information interface.

5 PEERHOOD IMPLEMENTATION WITH QT FRAMEWORK

In the next, an implementation of the PeerHood middleware concept based on the Qt framework is described (PeerHood2). The existing PeerHood (PeerHood1) implementation is used as a base for the new implementation. At the beginning of the project, it was decided that target is not just rewrite PeerHood1 again using the Qt framework. Without limitations and dependencies of existing protocols, – which were not well documented –enhance of the existing PeerHood implementation can be done better without problems. That decision enables doing things way of Qt and make PeerHood2 to be cross-platform middleware. Thus, PeerHood2 do not need to be compatible with the PeerHood1 implementation.

5.1 Features Implemented

In the chapter of PeerHood communication middleware introduction, the key features of the PeerHood concept were presented. Most of these requirements were implemented into PeerHood1. In addition, most of these were implemented to the new PeerHood2. The detailed comparison table of PeerHood implementations can be found from Appendix 2. Represented features can be roughly divided into two groups: group for the PeerHood API features and group for PeerHood internal – less user-centric – features.

Basis for a new PeerHood API functionality was to keep API as much same as it was in the previous PeerHood API. For that reason, there were no big changes in the functionality of the new PeerHood API. The biggest changes for the API come from use of the signals and slots mechanism to replace formerly used call back mechanism. Additionally, the new PeerHood API was harmonized to use a Qt coding conventions [41, 42]. Besides, the whole PeerHood implementation was put inside of a PH namespace to avoid name conflicts between PeerHood and third party applications. The new PeerHood API can be found from Appendix 3.

With the PeerHood2 interface, third parties applications can publish own services and as well discover other services and devices from network neighborhood. The PeerHood2

support as well connection to services with an abstracted connection. The connection abstraction enables application developer to use service connection in network transparent manner. The data transmission is not depending on the PeerHood implementation after connection is established and moved to PeerHood client. Moreover, the PeerHood API enables monitoring remote device actively or by using signal strength of connection.

Event interface for PeerHood1 was implemented using Observer design pattern [14], where callback interface instance was provided to PeerHood. The PeerHood uses provided callback instance for notifying events to client. For callback mechanism, the Qt framework provides solution, which is better suitable for particular application. The signals and slots mechanism are used for provide events to PeerHood clients.

The second key feature group is PeerHood internal features. Most of these features were implemented into the PeerHood2 as well. One of missing functionalities is the user control of PeerHood daemon, which was not implemented in PeerHood1 either. In addition, the network roaming functionality is not implemented.

The common structure of the PeerHood2 is a pretty much same than in the PeerHood1 implementation. Accordingly, use of network plugins in PeerHood2 is based on dynamic plugins, which are loaded at application startup. The connection network plugin interface provides channel for receiving events from a connection manager. With the connection manager, different networks and network plugins can be controlled.

Communication concurrency in the PeerHood2 is relying on the Qt event system, hence multithreading is not used. Concurrency provided by the Qt event system seems to be enough for the PeerHood use, however design of the new PeerHood2 is implemented in such manner that multithreading can be taken easily into use if needed. The decision of not to use multiple threads is made for resource saving point of view. Furthermore, multiple threads increases complexity of the system and make debugging much harder.

5.2 Architecture

Mostly the existing PeerHood implementation is event-driven system using event loop to receive events and deliver them to correct handler. Both PeerHood daemon and library uses infinite loops for handle incoming events and timed operations. The PeerHood library can be divided into two different parts. One part handles incoming service connections and another part provides interface for all PeerHood functionalities. Incoming service connections are handled in reactive manner based on events from network. Second part handles PeerHood API functions mostly communicating with the daemon and establishing connections to other services.

The Qt applications uses a same kind event loop systems than the PeerHood1 uses. Therefore, taking the Qt framework in use for the PeerHood does not cause major changes of the PeerHood design. Hence, the existing PeerHood1 design can be utilized. The biggest changes of the PeerHood structure comes from use of the signals and slots mechanism, which can help increase flexibility of the system. With use of the signals and slots mechanism, the use of Observer design pattern can be removed. Consequently, component coupling can be decreased with utilizing signal and slot mechanism in design.

As a basic structure of the PeerHood2 does not differ much from the PeerHood1, the use of daemon, library and network plugins were obvious. Moreover, PeerHood functionalities were separated to independent dynamic libraries. By use of independent dynamic libraries, a better reusability and changeability can be reached. In the PeerHood1 a lot of code is compiled to both, *peerhoodd* executable and *peerhood* dynamic library. New components for the PeerHood are common, settings and register. In the Figure 6 are shown PeerHood2 components and their relations.

The register component is an independent component for storing information of current devices on network neighborhood and information of locally registered services. The register interface enable insert, remove and search devices and services from the register.

The PeerHood settings can be handled with the settings component. The settings component can be used from every component, if access of common PeerHood settings information is needed. The settings component is based on use of the Qt settings class and therefore settings can be accessed from different processes easily without problems. However, this PeerHood component is not published to use of third parties. For third parties, the library component provide an own settings interface.

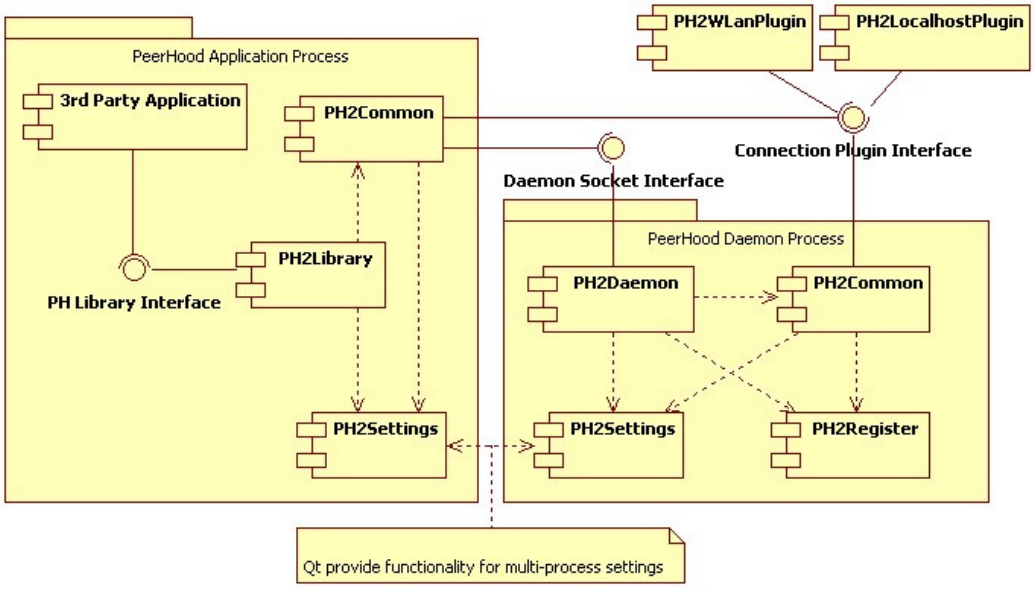


Figure 6. Component diagram for a new PeerHood structure

The Qt's signals and slots mechanism enables to remove a listener framework [43] and listener plugins from the PeerHood1 implementation. The listener framework was implemented to provide events of system changes. In the PeerHood2 system, events can be received with different Qt components via signals.

5.2.1 PeerHood Common

All general utility classes and common functionalities needed by PeerHood library and daemon are included in the common component. The common component provides functionality for plugin management, PeerHood data streaming to I/O devices and data containers for service and device information. The most of the PeerHood core services are included into common library.

Connection Manager

One of the most important functionalities in the PeerHood common component is a connection manager. The connection manager is responsible to load and control different networks with available network plugins. The connection manager is needed in both PeerHood daemon and library processes, due to both needs network connectivity in different networks. The connection manager controls plugins which are loaded. Loaded plugins depends of information what network types the device can handle. Network plugin is not loaded, if the network is not supported by the device. However, based by settings, some of plugins can force to be loaded. In addition, loaded plugins can be limited only for restricted plugins. The connection manager uses Bearer management API from the Qt Mobility extension to resolve available network configurations. Moreover, the connection manager receives events of added, removed or changed network configurations. In Figure 7 are shown classes related to the connection manager.

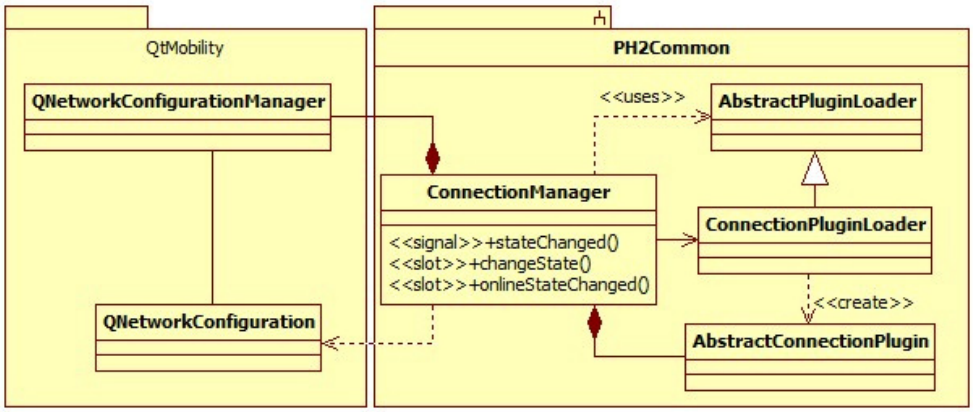


Figure 7. Classes related to Connection Manager

A ConnectionManager interface provides notifications of plugin state changes and ability to control system connectivity. In addition, the ConnectionManager provides a way to control some specific network types by events. The ConnectionManager interface is exported interface for use of other libraries. The ConnectionManager uses a Private Implementation (Pimpl) idiom [44] to enable better changeability without taking care of binary compatibility issues.

In the connection manager plugins are loaded with a ConnectionPluginLoader class. The ConnectionPluginLoader class is based on use of the QPluginLoader. Most of plugin loading functionalities are abstracted to AbstractPluginLoader class, which uses virtual functions to configure where plugins are loaded. Finally when plugins are loaded, a real plugin instance accepting is requested from concrete subclass. The subclass of the AbstractPluginLoader handles plugin casting to correct plugin type. If plugin type is not correct the instance is ignored.

After plugins are loaded with the ConnectionPluginLoader, the ownership of the plugin instances is taken by the ConnectionManager. The ConnectionManager can send different kind events to concrete plugin instance. These events can be like notifications of low battery level or request to going offline state.

Common Data Transmission

Service and device data transmission is unified in the PeerHood2 implementation. The PeerHood common component provides classes for sending and receiving device and service information without knowing how data transmission is actually done. With Reader-Writer classes, a single instances or list of instances can be send and receive in I/O devices. The Reader-Writer classes are used to abstract real communication between remote peer. The Reader-Writer classes are abstracted into level of the QIODevice. Hence, used device where data is sent or read can be process, socket or even file. In Figure 8 are shown Reader-Writer classes and a DataManager class.

The DataManager class provides an interface for sending information of registered data. The data can be locally registered services or discovered devices. In Addition, the DataManager enable common way to read information of remote device and send local device information to remote peers. Purpose of the DataManager is to provide a safe way to read information from register and send it to remote peer. This way, the data protection of registered data can be done better. The DataManager safely locks used data for read to avoid simultaneous data usage problems which might happen when multiple threads are used.

The common data transmission is used all over the new PeerHood implementation. Therefore services and devices are serialized to stream in same way when information is shared between daemon and client or between remote devices during network advertising. This way transmission logic is only for Reader-Writer classes and data containers. The data containers implements data serialization and deserialization from stream and Reader-Writer classes contains information of how single or list of data is transferred.

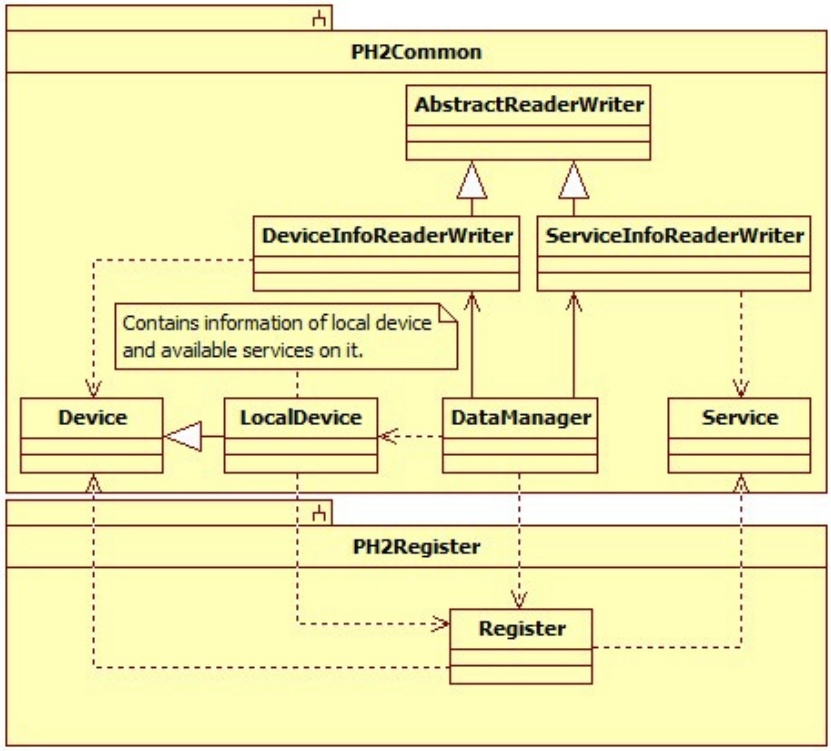


Figure 8. DataManager and Reader-Writer classes

Daemon Client

PeerHood daemon access is provided by the DaemonClient. The DaemonClient is part of the PeerHood common library. Purpose of the DaemonClient is to hide a real communication between client and PeerHood daemon. The DaemonClient provides interface of how daemon can serving its clients. Like Figure 9 shows there are two related classes for the DaemonClient.

The DaemonClient utilizes Acceptor-Connector design pattern [33]. At the beginning a DaemonConnector established a connection to the daemon. After that, the DaemonConnector initiates a DaemonClientService and provide instance of it to the DaemonClient. Using the DaemonClientService the DaemonClient can communicate with the daemon.

The Acceptor-Connector design pattern separates the connection handling from connection establishment. As a consequence of that, all communication logic is only in the DaemonClientService, which is easy to modify or replaced. Current implementation contains only ability to request something from the daemon, however in the future the DaemonClientService can also provide events from the daemon.

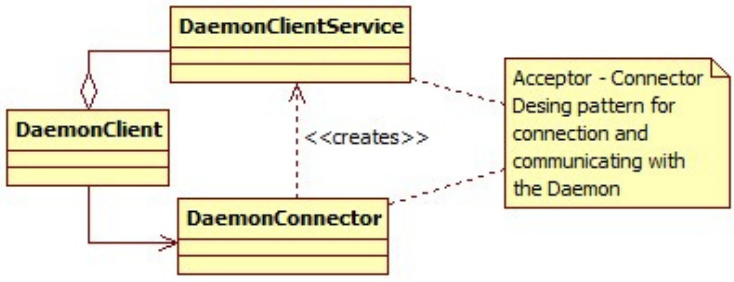


Figure 9. Classes related to DaemonClient

Factory

A Factory class utilizes common Factory design pattern [14]. With the Factory class, concrete implementations of provided abstract interface classes can be created. There are available abstract interfaces for remote device pinging, remote device monitoring, network advertising and for connectivity. Each network plugins register a creator instance in the Factory. The creator instance is used to create a concrete instance. If network plugin creator does not support functionality, it can return a null instance. The concrete creator registration and instance creation sequence are shown in Figure 10.

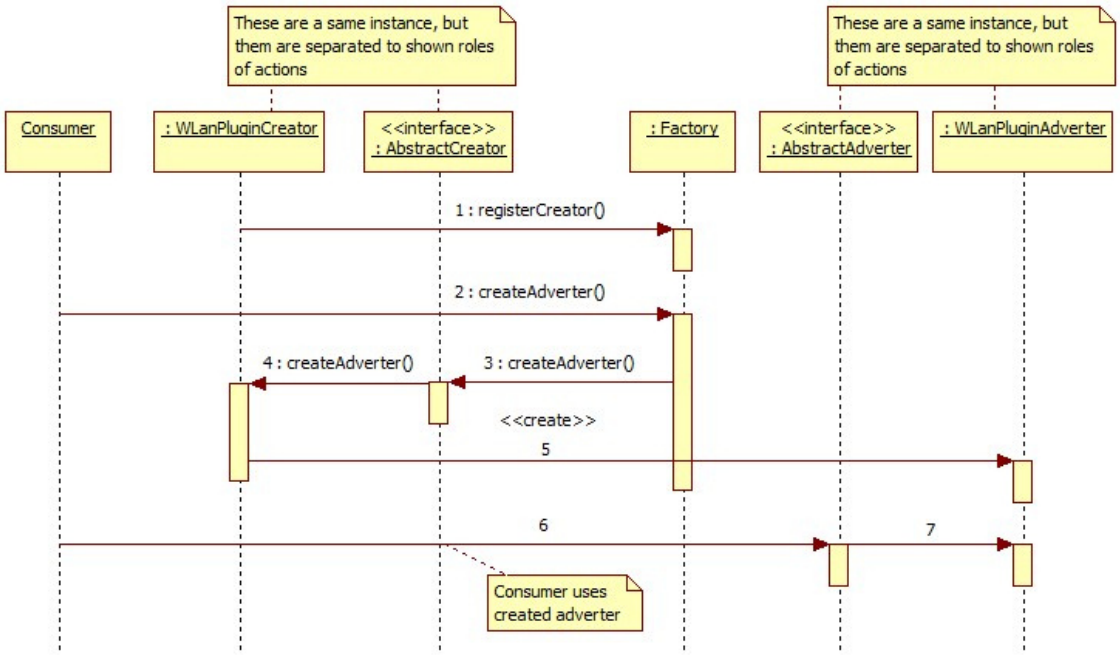


Figure 10. Sequence diagram of concrete creator and instance creation

5.2.2 PeerHood Daemon

Compared to the PeerHood1 daemon implementation in the PeerHood2 daemon implementation has been changed a lot. In the PeerHood1, daemon implementation was one a huge CDaemon class, which contains all daemon related functionalities. In the PeerHood2 implementation, that class has been split to several classes. The main Daemon class uses these classes by aggregation. Internal structure of the daemon and class relations is shown in the Figure 11.

The daemon has two fundamental functions, which are it responsibilities. The daemon is responsible for publish information to other devices and discover information from other devices in the network neighborhood. Likewise, the daemon provide interface for clients to request information maintained by the daemon. In Addition, PeerHood clients can insert and remove own services to be published.

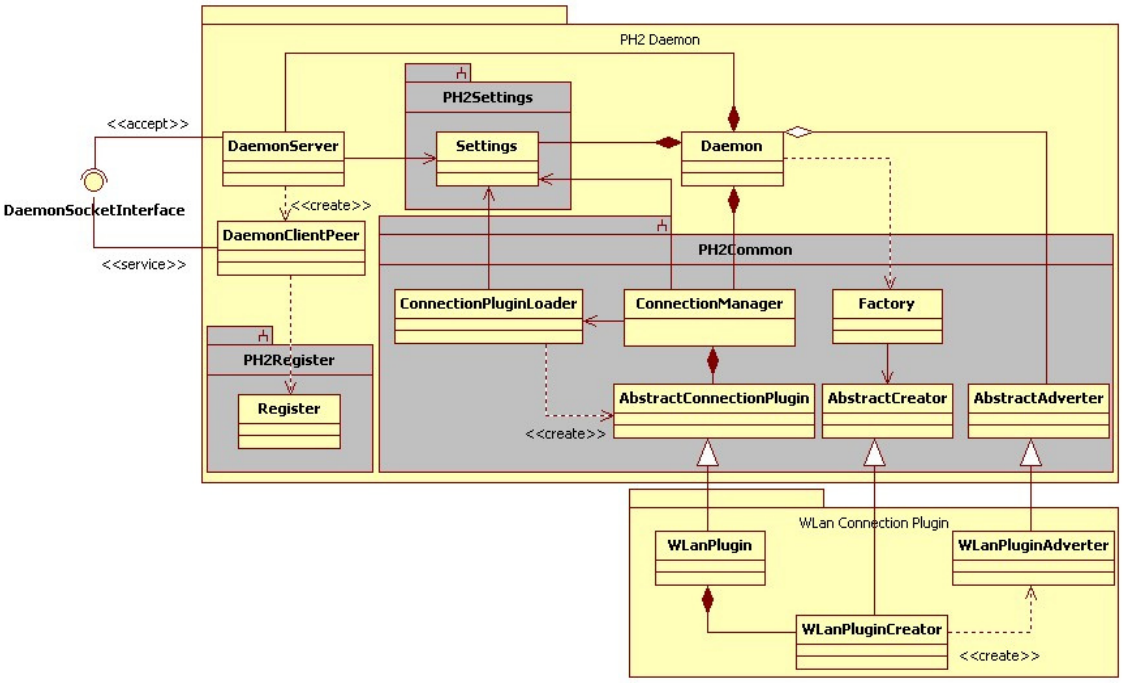


Figure 11. Classes and their relations in the PeerHood daemon

Actually, either of the PeerHood1 and the PeerHood2 daemon implementations does not contain device discovery and publish information. That functionality is divided into each network plugins. Only responsibility of the daemon is to start network advertising functionality for each network plugins. After advertising is started, plugins handle device information publishing and discovering. The daemon constructs advertisers for the each network plugins with the Factory provided by the common library.

Second daemon function is to provide interface for daemon clients. The clients must be able to fetch information of detected devices and available services. In addition, clients must be able to register services and remove registered services.

The daemon provides a local socket interface for clients. For handling incoming connections, the daemon utilizes Acceptor-Connector design pattern as well. A DaemonServer is used to listening incoming client connections. When a new connection arrives the DaemonServer accept the connection and create instance of a DaemonClientPeer class. After the DaemonClientPeer instance is done, the

DaemonServer shift connection responsible to the created DaemonClientPeer instance. The DaemonClientPeer handles connection until connection is closed.

5.2.3 PeerHood Library

Like in the daemon implementation, also the library implementation was divided to several smaller classes. Purpose of the creating smaller pieces is to create classes with clear independent functionality. The PeerHood class is a Façade [14] for the whole PeerHood system. That class provides service connectivity functionality, device monitoring functionality and functionality provided by the PeerHood Daemon interface. Figure 12 shows classes related to the PeerHood API class. For keeping figure clear enough private implementation class for the PeerHood class is not drawn. However, The PeerHood API utilizes Private implementation idioms for the PeerHood API as well.

Service Connection

The PeerHood library handles connection establishment between services. The connection establishment consist accepting incoming service connections and initiate service connections to other services locally or remotely. The PeerHood library uses the Acceptor-Connector design pattern for this purpose as well.

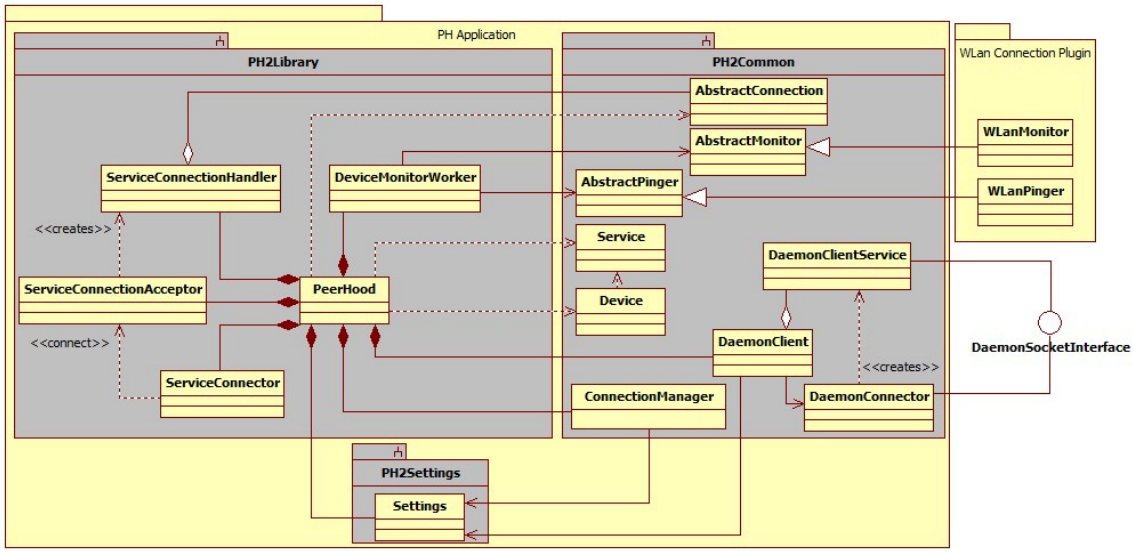


Figure 12. Classes and their relation in the PeerHood application

A class ServiceConnectionAcceptor is responsible for accepting incoming connections. The ServiceConnectionAcceptor creates connection instances for all available network types and start each of them to listening incoming connections. When an incoming connection arrives, the ServiceConnectionAcceptor creates an instance of a ServiceConnectionHandler class. The ServiceConnectionHandler initiates connection and notify PeerHood API client with *newConnection* signal. Execution sequence for incoming service connection is shown in Figure 13.

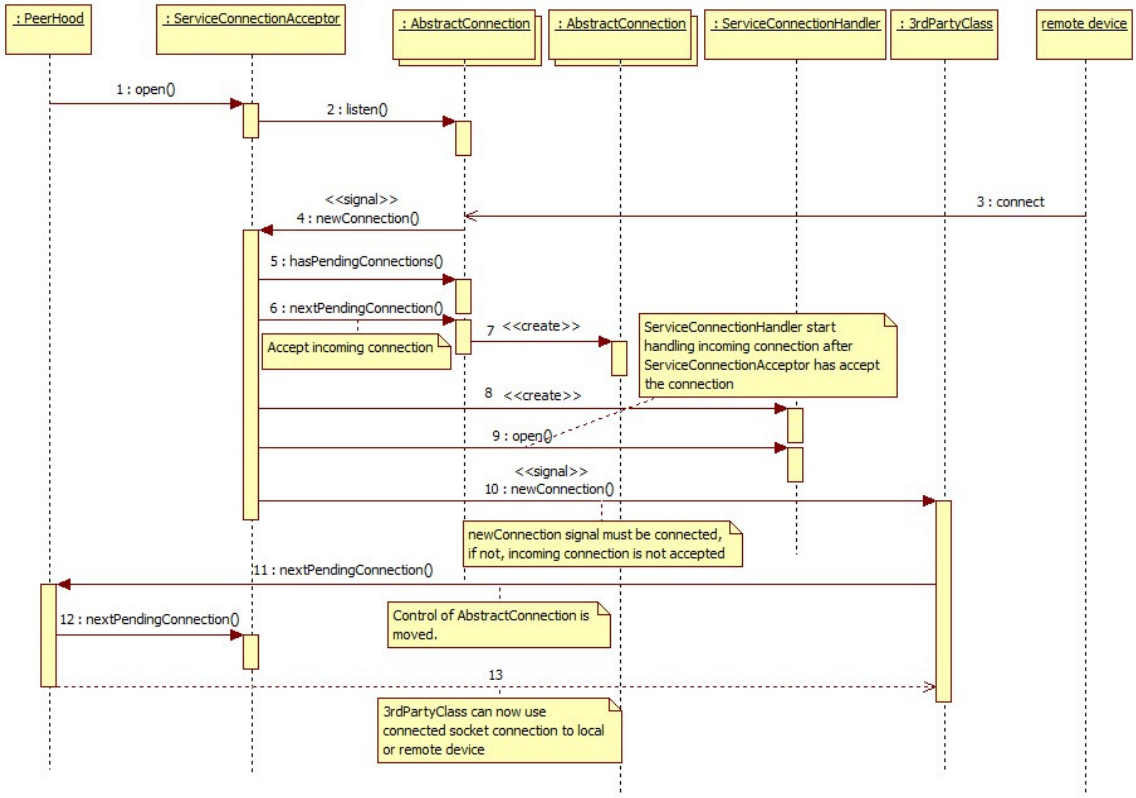


Figure 13. Incoming service connection

For connecting to another service a ServiceConnector class can be used. The ServiceConnector provide only simple interface for initiating connection to wanted service. Internally the ServiceConnector first establish a connection to another service located on local or remote device. After connection is established the ServiceConnector initialized the connection by sending some additional information. Sequence of Service connection initialization is shown in Figure 14.

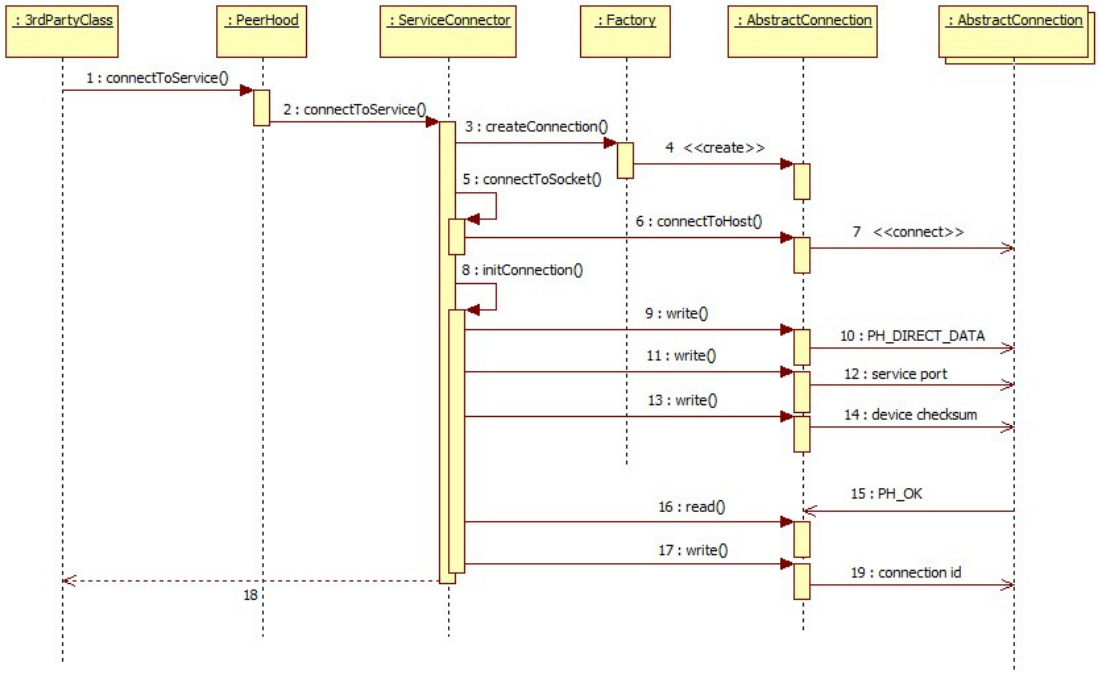


Figure 14. Service connection initialization

Device Monitoring

The PeerHood contains two different kind device monitoring functions. These are device monitoring based on active device pinging and device monitoring by signal strength between devices. For feasible device signal monitoring the direct connection between devices is required. Otherwise, signal monitoring is done of connection between device and access point, like connection between device and WLAN router.

The both monitoring options are included into a DeviceMonitorWorker class. In the PeerHood1 monitoring was done in separate threads and for that reason both monitors are included the DeviceMonitorWorker, hence monitoring functionalities are easy to move to run in another thread if needed.

An abstract signal monitor interface was built insight that the Qt Mobility extension provides functionality to receive events of signal strength changes. Signal strength events come from the Qt Mobility without any active polling by the PeerHood. As opposite to signal monitoring, the active monitoring requires operate pinging within interval. The QTimer is used for active monitoring timing.

5.2.4 PeerHood Network Plugins

Like the PeerHood1, also the PeerHood2 contains dynamic plugin extension system for including different network specific implementations. In the current PeerHood2, only plugins for local host and WLAN connectivity are implemented. The local host network plugin provides only ability to create local host based connection instances, in that other functionalities are not supported on local host. In addition, the WLAN network plugin contain all functionalities, which network plugins can provide for the PeerHood.

The plugin system and interfaces are very similar in the PeerHood2 than in the PeerHood1 implementation. However, some things are done a bit different way than in PeerHood1. Two major changes to plugins system are a new interface for service advertising and capability to receive and send events by plugin interface. Classes related to WLAN network extension are shown in Figure 15.

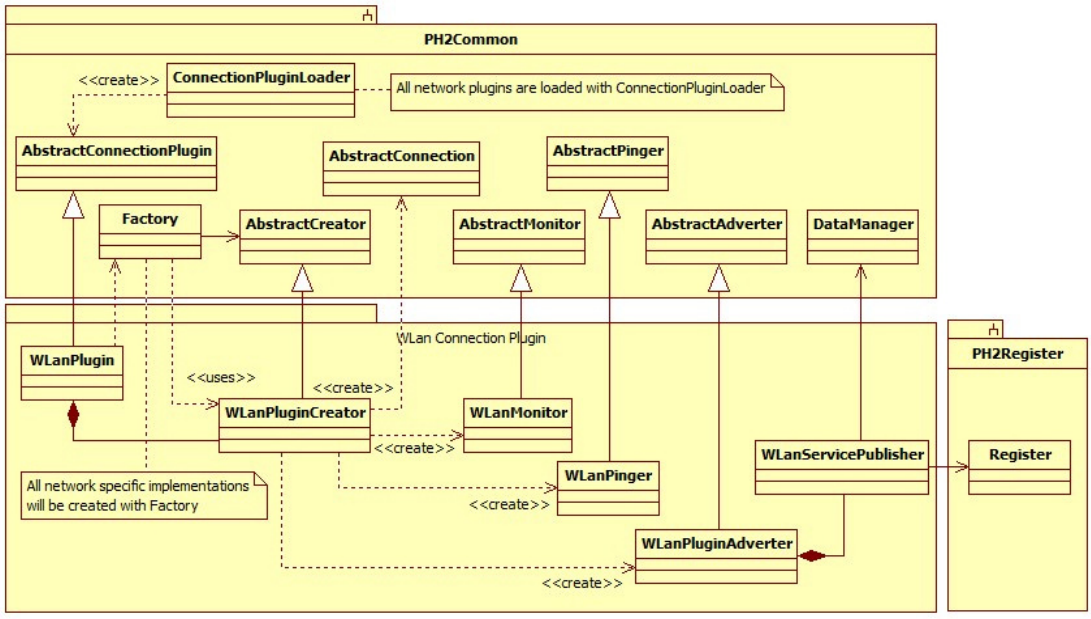


Figure 15. Class relations in WLAN connection plugin

Advertising Interface

The network advertising control functionality was included in the plugin interface in the PeerHood1 implementation. The new PeerHood2 implements an own AbstractAdverter interface. The AbstractAdverter can be used to manage network advertising. Like

other abstract interfaces for network plugin the AbstractAdverter can be created via Factory class. Even though, advertise controlling is moved to own interface it provides same functionalities as earlier.

Network Plugin Events

Network plugins interface – an AbstractConnectionPlugin – is extended to have ability to receive and send events. The AbstractConnectionPlugin interface utilizes the signals and slots mechanism for events. The connection manager can notify plugins with the event system. Events for plugins can be notifications of common events like low battery or control events like request to go offline state. Thus, the ConnectionManager can control all plugins with events. Furthermore, the network plugin can notify others if it changes state. It is notably that action for events depend plugin implementation, as a result, plugin implementation can ignore provided request, like going in battery saving mode.

5.2.5 PeerHood Applications

The biggest change for the PeerHood API using is that the new PeerHood API requires use of the Qt framework also. If use of the Qt framework is not possible in third party application, a C++ wrapper for the PeerHood API can be implemented into PeerHood. Other impacts or limitations for the third party applications the new PeerHood API does not cause.

The previous PeerHood implementation rejects incoming service connections if callback instance was not given at PeerHood init. The PeerHood2 uses similar model by recognizing is *newConnection* signal connected to any slot. If signal is not connected, the PeerHood rejects incoming service connection requests, as there is no one to accept incoming connections.

5.3 Improvement Ideas

Using the Qt framework enables ability to extend the PeerHood with new uses cases. The Qt provides several APIs to get notifications of changes in the system and device.

With the event system for network plugins, the ConnectionManager can be extended to control network plugins, like reduce power consumption when device is running out of battery. In addition, network plugins must be implemented to change their behavior when events are provided by the ConnectionManager.

Second improvement idea is related to device detection and information sharing in network neighborhood. Currently, in both PeerHood1 and PeerHood2 responsibility of device detection and information sharing is in network plugins. With the connection abstraction, these functionalities can be centralized to daemon. In that model device, detection logic is only in one place and there is no need to implement that in all network plugins. Network plugins can be used to parameterized inquiries and other needed things based on network type or some other details. Centralized model decreases also dependencies of network plugins.

The PeerHood1 contains support for the Bluetooth network as well. In the PeerHood2, the Bluetooth was leave out of scope, because of the Qt framework does not provide needed functionalities for that. Hence, Bluetooth implementation must be implemented in the platform specific way. The Bluetooth plugin implemented in the PeerHood1 can be used in the PeerHood2 with a small modifications and integration to existing plugin system.

6 EVALUATION

In this chapter, the new PeerHood2 implementation is evaluated. Evaluation is done with using explained McCall's quality factors. In addition, the Peerhood2 implementation is compared with the PeerHood1 implementation to see impacts of the Qt framework into PeerHood implementation.

6.1 Test Environment

All tests were executed on 10.04 Ubuntu with 2.6.32-24-generic kernel version running on VMware player 3.0 virtual machine. As a host platform, Asus EeePC (Intel Atom N450 @ 1,66GHz, 1Gt RAM memory) with 32bit Windows 7 Starter operating system was used. The virtual player was configured to have 640 MB RAM memory.

6.1.1 PeerHood Configuration

Both PeerHood implementations were measured as same configuration as can be. The test configuration consist network plugins for localhost and WLAN networking. The WLAN advertising interval is set to one second in both PeerHood implementations. In addition, the old PeerHood implementation contains feature for customizing what information the PeerHood shares between devices. This feature is set to be aligning with the new PeerHood implementation, which share service and device information between devices.

The PeerHood2 is built with Qt 4.6.3 version and 1.0.2 version of the Qt Mobility extension APIs are used as well. The Qt framework and the Qt Mobility are compiled to Linux environment with default configuration. All builds – including PeerHood1 – are done with the GNU C++ compiler. The used PeerHood1 version is SVN revision 170 of public PeerHood1 SVN repository [45]. The PeerHood2 version can be found from Gitorious [46].

6.1.2 Active – Passive Client Test Set

Purpose of an Active-Passive test set is to cover common PeerHood actions in a same test set. The test set is limited to local connectivity to avoid disruption caused by network transmission. The proposed test set contains functionalities of service registration, service resolving and connection to provided service. Figure 16 shows the test set and steps of the test. The test set is based on active – passive PeerHood client pairs. All active and passive clients are running in own processes.

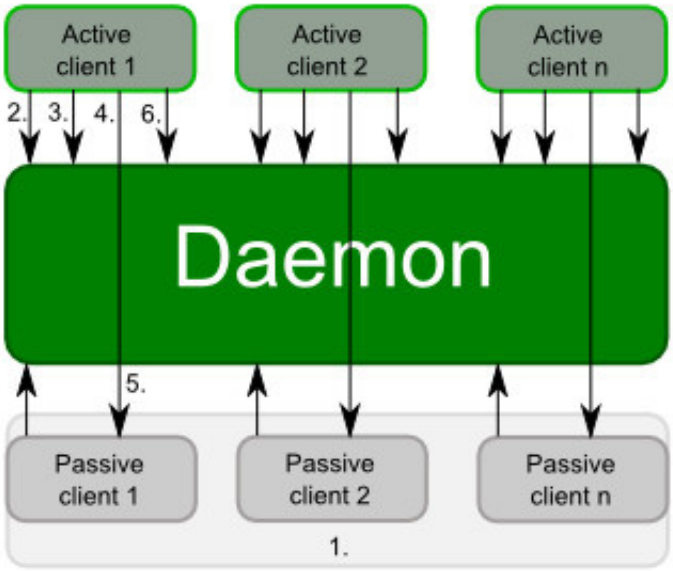


Figure 16. Used PeerHood test set

First, for each active clients must start a passive client (step 1). For each passive client registers a service, which active client pair is going to use. After all, when passive clients are started, active clients can be started. The active clients execute the following test set in five seconds interval (Figure 16):

2. Register a new service
3. Read all services and verify that at least two services are found. These two services must be service registered by the passive client pair and second must be service registered by self.
4. Find and connect to service published by the passive client pair

5. Passive client accept a service connection. After connection is made. The active client verifies result of connection and disconnects the created connection.
6. Finally active client unregister service which was registered in the step 2.

All executed test steps are verified by the active client. If some step fails, the execution of the active client is interrupted and the execution time is logged.

6.2 Maintainability

For the maintainability, the static code analysis is used to give some overview of differences between the PeerHood1 and the PeerHood2 implementations. The static code analysis is done with SourceMonitor 2.6.3 application [47], which is freeware source code analyzing tool. For the PeerHood1 implementation, only main PeerHood, localhost plugin and WLAN plugin are analyzed. Hence, the both PeerHood1 and PeerHood2 are comparable together. The Bluetooth and GPRS plugins are excluded because the lack of functionalities in PeerHood2.

Static code analysis results are divided into two different groups. The first group is a quantitative metrics of PeerHood implementations and second group is more qualitative metrics of PeerHood implementations. Figure 17 shows quantitative metrics of PeerHood implementations. The amount of Files, Lines of Code (LOC) and class definition metrics are obvious and does not need explanation, nevertheless statements, branches and functions may need.

The statements metrics is amount of computational statements in the source code [47]. These statements includes all C++ statements including branches such *if* statements and loops such *for* and *while*.

Statements that cause break for sequence execution of code are counted in the branches metric [47]. The branches metric is percentage value of the all statements. The branch statements are for example *for, while, if* and *switch case* statements.

Functions metric is total amount of functions declared out of class definitions [47]. This metric includes all functions although function is defined in the source code file scope with the static definition to be out of global namespace. The functions metric counts the main function as well. The PeerHood1 contains C API for the PeerHood as well, which causes that big difference between amount of functions in the PeerHood1 and the PeerHood2 implementations.

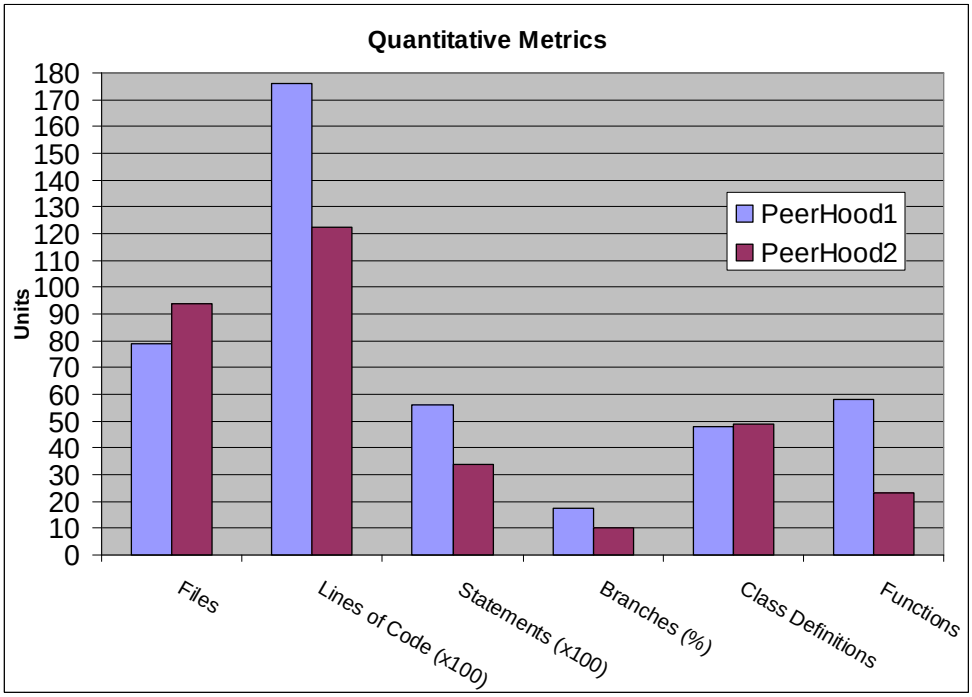
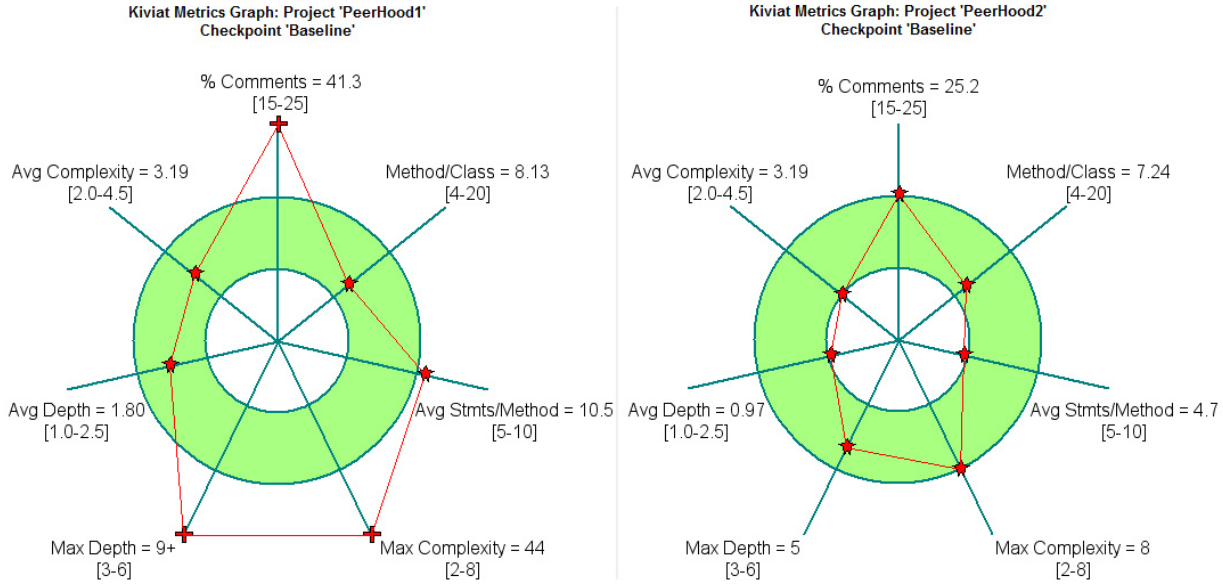


Figure 17. Quantitative metrics for both PeerHood implementations

Especially the difference of lines of code is remarkable in maintainability point of view. Evidently, the quantitative metrics is nothing without analyzing quality of code. Code quality metrics are shown in Figure 18. Kiviat graphs for the PeerHood1 (A) and the PeerHood2 (B) contains information such a code complexity and a code depth. Moreover, average amount of statements in methods and code comments are available.

In the Figure 18 the green area and values inside of angle brackets are preferred values of each metrics [47]. The exact metric values of the implementation are shown in each axis description. Except, the maximum depth value is limited to nine and all statements at the deeper level are counted as in depth nine [47].



A) B)

Figure 18. Static code analysis metrics for A) PeerHood1 and B) PeerHood2

The depth metrics describes nested blocks in a method or a function. The nested blocks are usually introduced with execution control statements such if and loops statements [47]. Hence, depth metrics are related to conditions in a code. More conditions in code make code harder to read. Namespace blocks are not calculated into depth metrics.

The code complexity metrics are calculated from code condition options. For each conditional statements increases execution paths in a method or a function. The code complexity is calculated from these execution paths, which increases the complexity of method [47]. The code complexity calculation is based on definition by Steven McConnell presented in his book Code Complete [48].

Combining lines of code with the code complexity give a good overview of code to be maintained. Code with a high complexity value tends to be difficult to maintain and usually has high defect density [16]. Furthermore, there are relation between complexity and software reliability as well [17].

All differences of quantitative and qualitative metrics between PeerHood implementations are not only related to use of the Qt framework. A lot of code has been refactored during the new PeerHood implementation and some features like connection

roaming was dropped off the PeerHood2. However, the Qt framework provide a lot of functionalities to PeerHood, which helps to keep internal structures more simple and code less by self.

6.3 Reliability

In reliability testing the introduced active-passive client test set were used. The purpose of reliability test was to verify how all PeerHood components work in long running operation. Time frame for test was set to 12 hours. The PeerHood system should work that time without any abnormal behavior. The active client verifies all operations and terminates client execution if some operation failed. The active-passive client tests set were run with five client pairs and all executions expected to last 12 hours.

Figure 19 shows results of the reliability test. The PeerHood1 implementation fails after 85 minutes to socket assertion error. For that reason, PeerHood1 implementation was tested three times. Every time all clients fail in that same runtime. It seems that some sockets are not closed proper way in PeerHood1 implementation and finally socket creation fails. The PeerHood2 ran 12 hours without any abnormal behavior. The test execution was stopped after 12 hours was elapsed.

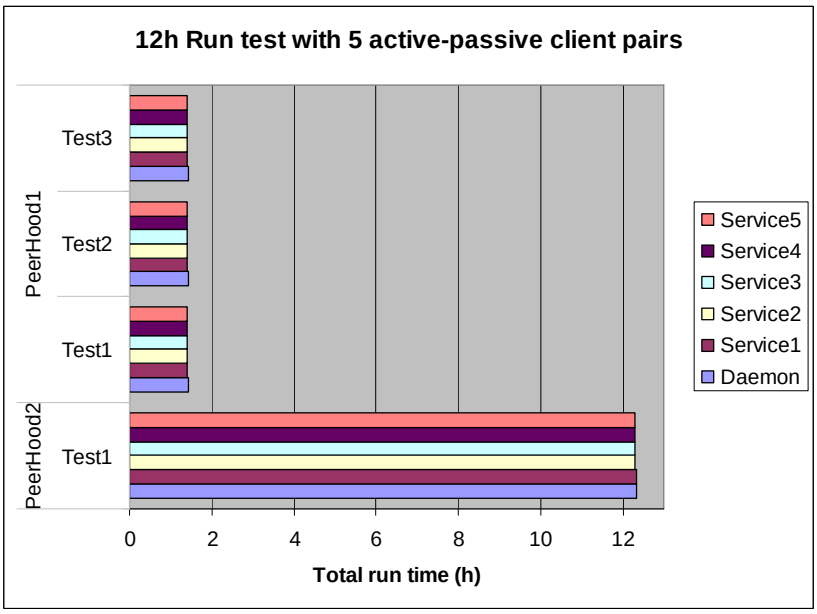


Figure 19. Reliability test results

Deeper analysis of the PeerHood1 failure in the reliability testing was not done. The failure was very deterministic and it occurs in every test run almost at the same time. As the previous chapter states, there are relations of the code reliability and code complexity, which can be partially seen in this reliability test as well.

6.4 Efficiency

PeerHood efficiency is evaluated with measuring resource usage of both PeerHood implementations. Memory usage and CPU usage of both implementations are measured and compared in following subchapters.

6.4.1 Memory Usages

PeerHood implementations memory usages were analyzed with two different memory usage tools to get better reliability for the results. The memory measurements were done on Linux platform, which uses virtual memory. The virtual memory can be a bit challenging when measuring memory usage, due the fact that memory areas can be reused between multiple processes [49].

Memory usage is measured for all related executables in the active-passive test set. The test set consist executables for PeerHood daemon, passive client and active client. In the memory usage tests, the test set was executed with the five active-passive client pairs in duration of ten minutes. All memory measurements were done from application start to application close.

Massif

PeerHood memory usages are measured with a Massif tool, which is heap profiler tool in the Valgrind [50]. Output of the Massif is visualized with a MassifG tool [51]. The Massif can be used for measuring how much heap memory application consumes. In addition, the Massif can measure use of stack memory and extra bytes of heap allocation. Extra bytes are allocated in book-keeping and alignment purposes [50]. Moreover, the Massif can expose memory leaks on application runtime. The runtime

memory leaks cannot be determined with regular memory leak tools if application cleans allocated memory correctly at the application exit.

In Figure 20 are shown heap and stack memory usage of the PeerHood1 daemon and in Figure 21 are shown heap and stack usage of the PeerHood2 daemon. Difference of memory usage is significant between native application and the Qt based application. However, the memory usage level in the PeerHood2 is in usable range.

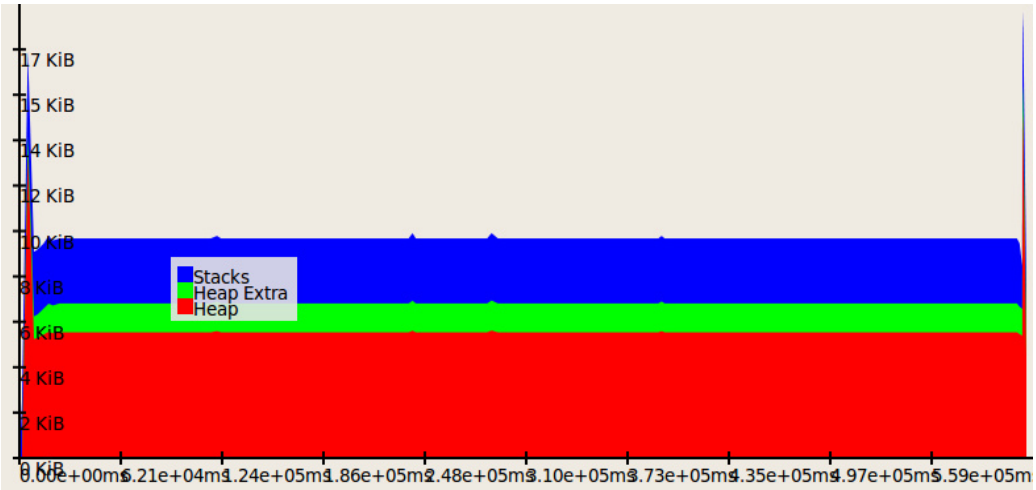


Figure 20. Memory usage in the PeerHood1 daemon, measured with the Massif tool

Figure 21. Memory usage in the PeerHood2 daemon, measured with the Massif tool

Between the active and the passive clients, the differences of memory usage were negligible and for that reason only memory usage of the passive clients are presented. In

Figure 22 is shown memory usage of the passive client which uses the PeerHood1 implementation. In Figure 23 is shown memory usage of the passive client with using the PeerHood2 implementation. As already mentioned, the Massif tool can point out application memory leaks, which can be seen in PeerHood1 based passive client. The difference of passive and active client memory usages was a bit larger than between daemon executables.

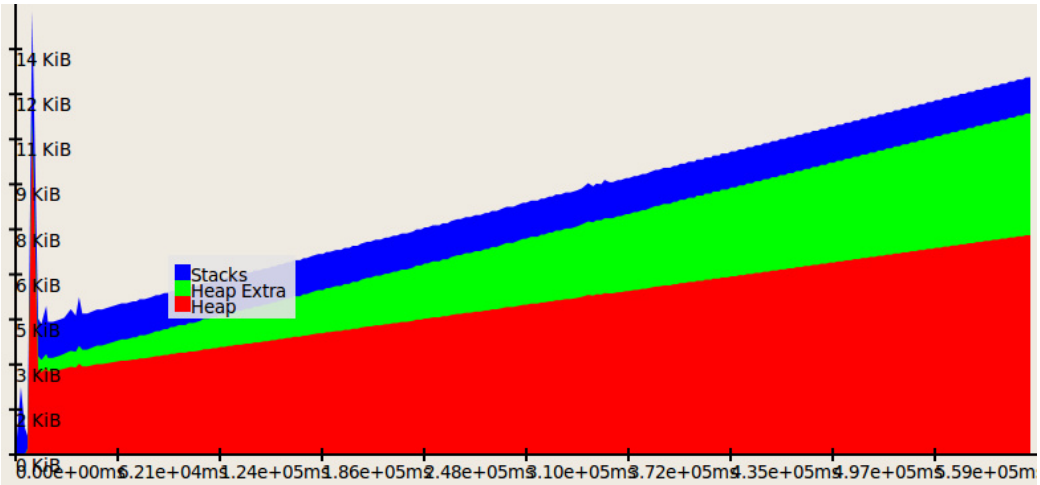


Figure 22. Memory usage of passive client using PeerHood1

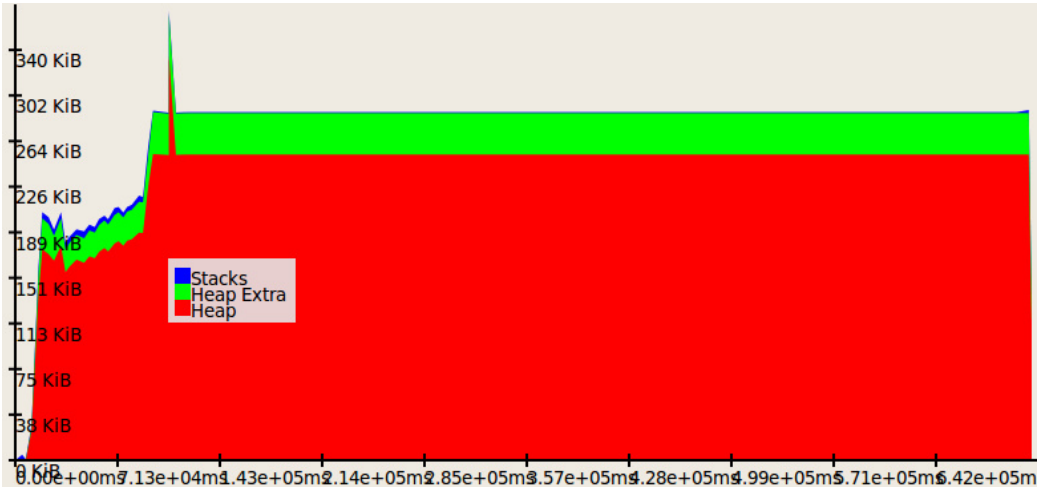


Figure 23. Memory usage of passive client using PeerHood2

Exmap

Second tool for memory usage measurements is Exmap tool [52]. The Exmap can provide more detailed information of used memory. The Exmap can take into account of

shared memory between processes by providing information of effective, mapped and resident memory in addition to heap memory. The result of Exmap memory measuring is a snapshot of currently running process. For that reason, memory usage results are average values of snapshots in time after minute, five minutes and ten minutes runtime.

In Figure 24 are shown different memory usage results. In all tests, resident and mapped memory was same amount of memory. For that reason, resident and mapped memory usages are grouped together in all related figures.

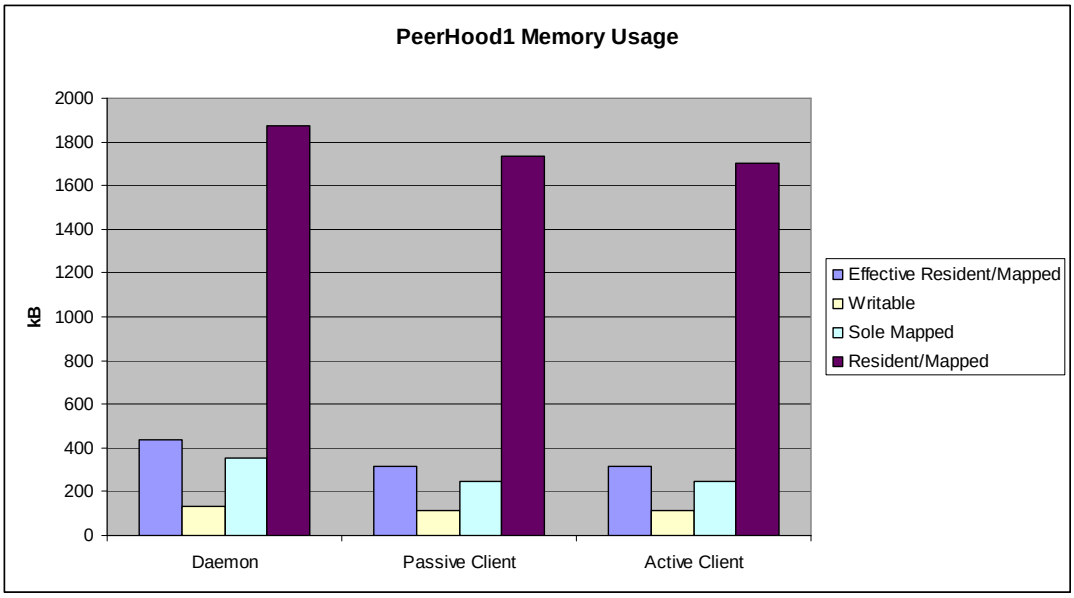


Figure 24. PeerHood1 memory usages measured with the Exmap

Resident memory size is amount of virtual address base mapped to physical RAM memory. The mapped memory size contains also amount of mapped virtual address base, except memory can be stored into physical RAM or into swap space [52]. Consequently, all mapped memory in all tests is located in RAM memory and swapping was not needed.

Effective resident and mapped memory metrics are more sophisticated metrics than regular resident and mapped memory metrics. In the effective metrics, shared memories are taken into account [52]. The Exmap recognizes shared pages and the page size are divided by all process which uses the memory page [49].

The sole mapped size is same as the mapped size, although it contains only pages, which are currently sole use by the process [52]. Hence, pages can be shared. In addition, writable memory size shows memory currently stored to pages, which are marked to be writable by the process.

Memory usages of the PeerHood2 implementation are shown in Figure 25. As the heap usage difference was major, the virtual memory usage in the PeerHood2 is also much more than in the PeerHood1 implementation.

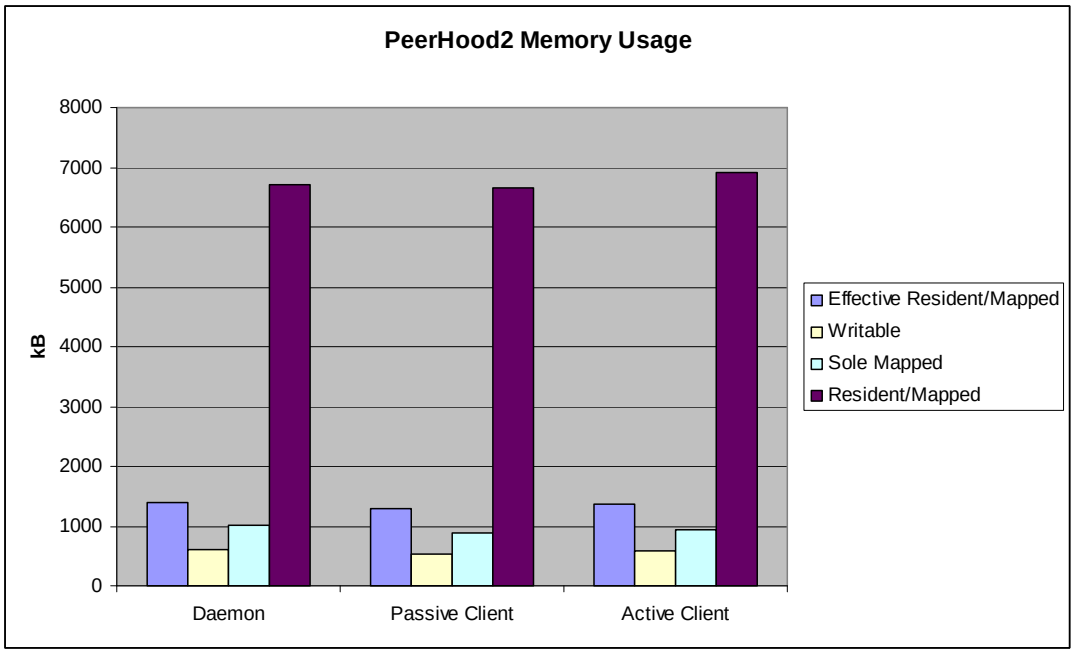
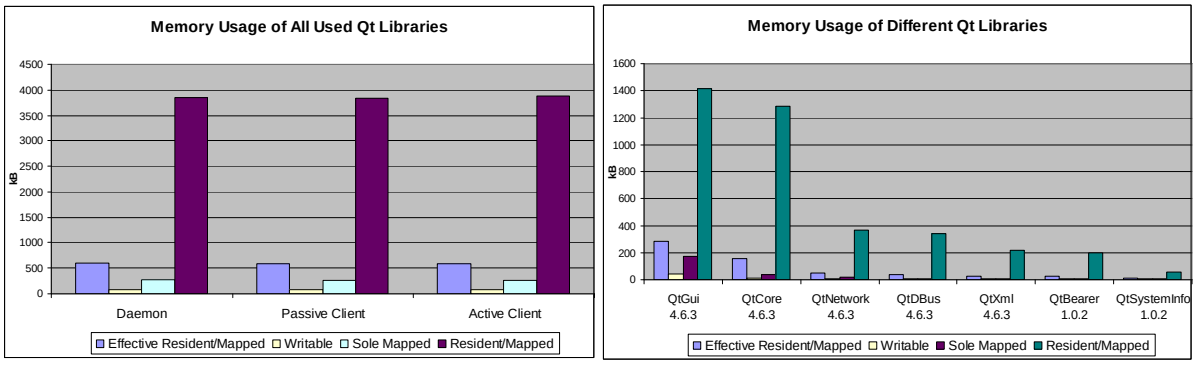


Figure 25. PeerHood2 memory usages measured with the Exmap

The Exmap can show detailed memory usage for each dynamic library and executable. Figure 26 shows separated portion of the Qt framework in the PeerHood2 memory consumption. In the Figure 26 A) memory usage of all Qt libraries are shown for each component: daemon, passive client and active client. In addition, Figure 26 B) contains detailed information of how much for each Qt component uses memory in the PeerHood2 daemon use.



A) B)

Figure 26. A) Proportion of Qt libraries in the PeerHood2 memory usage. B) Memory usage for each Qt library in the PeerHood daemon use.

With the Exmap, also the heap memory usages were measured. Results of heap memory usages can be seen in Figure 27. The results are comparable and align with the results from Massif measurements.

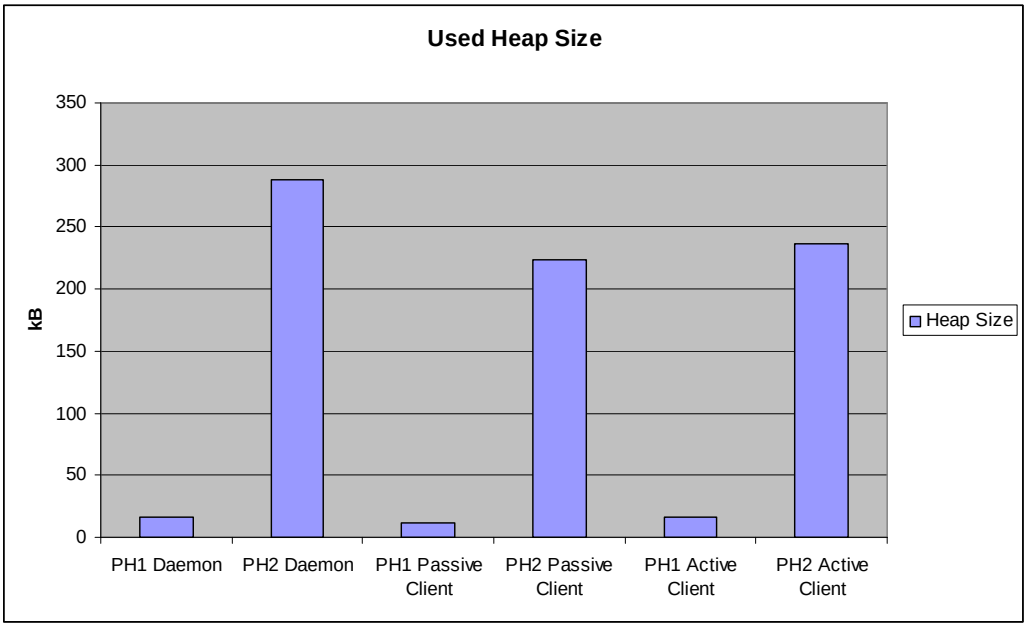


Figure 27. Heap memory usage differences between PeerHood1 and PeerHood2

6.4.2 CPU Usages

CPU usages of PeerHood implementations were measured with *time* command from Linux command line tools. The *time* shows used wall time – how long application actually were running – and also user and kernel times used by application under test.

CPU usages were measured for all processes related to active-passive test set. That includes the PeerHood daemon, passive client and active client. In Figure 28 are shown results of CPU usages. The test set was run on 10 minutes. After 10 minutes, applications were closed when application exit the time tool print result of applications system time usage.

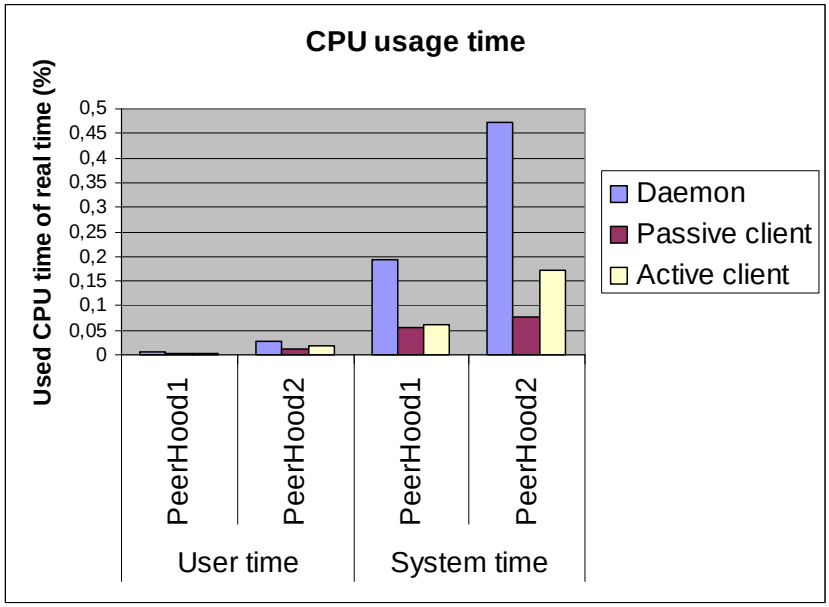


Figure 28. PeerHood CPU usages

Both PeerHood implementations uses only small amount of CPU time. However, differences in CPU usages between PeerHood implementations are significant. One explanation for CPU usage difference can be system monitoring used by the PeerHood2 implementation. In that, WLAN listener is not used in the PeerHood1 on desktop Linux environment.

6.5 Correctness

During the PeerHood2 implementation, API tests were implemented for verifying correctness of the implementation. API tests can be though as one part of extensive quality verification process. Moreover, more detailed evaluation and measuring correctness requires a full quality assurance process and detailed application requirements [12]. Larger quality assurance was not feasible in scope of this study.

Along with better correctness verification requires some test bed for testing device and service discovering from network neighborhood using different network technologies.

6.6 Testability

PeerHood2 was implemented with the set of API tests. Mostly these tests use APIs provided by the QTest test library. The QTest provides a good tool set for Qt based components testing. Test coverage's for the API tests were measured with the *gcov* [53] tool and output generated with *lcov* [54] tool. The results of the test coverage's are shown in Table 1.

Table 1. Test coverage results for the PeerHood2

Module	**Line Coverage (%)**	**Function Coverage (%)**
PH2Daemon	N/A	N/A
PH2Common	92,7	74,0
PH2Library	89,3	67,9
PH2Settings	98,7	82,6
PH2Register	94,0	82,6

Using code coverage tools with testing can provide valuable information to test developers. The test coverage data is good signal of how well and inclusive tests are. Hence, test coverage helps monitoring system testability – how well codes are covered by tests. However, the code coverage does not guarantee quality of tests and are correct things actually tested.

6.7 Flexibility

Like the existing PeerHood implementation also the new PeerHood implementation utilizes common design idioms and patterns for trying to keep PeerHood design flexible and changeable. The new PeerHood implementation also uses several independent shared libraries. With shared libraries, the middleware can be updated or changed by replacing only particular component or components without compiling entire

middleware. When using shared libraries and not compiling entire middleware a binary compatibility must guaranteed between components.

The Qt framework provides signals and slots mechanism, which can be used to increase system flexibility. With the signals and slots, components and classes can be loosely coupled. Consequently, dependencies of components can be decreased. Signals can be connected or disconnected without communication with related component. Therefore, application extension can be easier if signals and slots are used.

6.8 Usability

Even though PeerHood is API for third party applications, usability can be evaluated [13]. However, usability evaluation requires API usage evaluation with several application developers. The usability evaluation was out of scope of this thesis work.

The PeerHood usability is look after with keeping PeerHood documentation in good level and up to date. The PeerHood2 internal classes are documented with the Doxygen [55] to help understanding the internals of the PeerHood. In addition, PeerHood API documentation is generated with the Doxygen for the third party application developers.

The use of general data types and structures from the Qt framework decreases learning curve of the PeerHood API. The PeerHood1 contains own implementations of device and service lists. Instead of using common lists, the PeerHood1 API publishes own list structures, which third party application developers must learn as well.

6.9 Integrity

The PeerHood is research project and security of the PeerHood is not concentrated really much. The integrity issues were leave out of scope and data integrity of the PeerHood2 is keep as it is in the PeerHood1. The only significant change is use of dynamic library for the data registry. The use of dynamic library can enable replacement of data registry component. Replacing existing data component can enable some data manipulation and fake data can be provided. However, this is not probably a huge issue.

6.10 Portability

Maybe one of the biggest advantages of implementing the PeerHood on top of the Qt framework is get excellent portability across of different platforms and devices. The Qt framework itself provides a good portability across supported platforms. The most of the PeerHood is implemented as a platform independent way.

Some of PeerHood functionalities cannot be implemented with tools provided by the Qt framework. These parts need to be implemented with platform specific functions and will cause some porting effort when porting PeerHood from a platform to another. The Qt framework does not support for example using ICMP (Internet Control Message Protocol) protocol [56] which is used in active device monitoring. In addition, when implementing Bluetooth network plugin it must be done in platform specific implementation. In that, the Qt framework does not provide the Bluetooth controlling functionalities. Bluetooth device control is needed to publish PeerHood service to be found with Bluetooth Service Discovery Protocol (SDP) [57, 22].

When implementing a platform specific code into cross-platform code, use of abstraction is important to keep portability in good level. In addition, platform specific code is good to separate from other sources in the project, so that platform specific codes are only included into build when compiling to specific platform. Moreover, adding preprocessor conditions makes code harder to read and understand if a lot of platform specific code is embedded into generic code.

One option to use of ICMP based device monitoring is to implement device pinging on top of the UDP protocol. In this way, the device monitoring can be implemented in the cross-platform manner. In addition, in many networks the ICMP ping is blocked by the firewalls. Thus, the UDP based ping implementation can possible provide better availability of the active device monitoring functionality.

6.11 Reusability

The system architecture in the PeerHood2 is based on multiple dynamic components. The PeerHood consist two different kind dynamic component uses. Independent and common functions were separated in multiple components to be linked to the PeerHood in compile time. In the PeerHood1 some of common functionalities was built in the both PeerHood daemon executable and the PeerHood library, which increases the size of binary.

As one of the PeerHood requirements was, the PeerHood networks must be extendable with the plugins. These network plugins are implemented as runtime loadable dynamic libraries. Hence, a new network supports can be added dynamically into PeerHood. The network plugins can use services from linkable plugins provided by the PeerHood.

When using component based architecture with the Qt framework the common components can be shared across different platforms. In addition, utilizing the signals and slots mechanism components coupling can be reduced. The loose coupling between components enables flexible component reusability.

6.12 Interoperability

The PeerHood concept provides a good interoperability by itself with dynamically loaded plugins. With new plugins PeerHood interoperation of other systems can be extended. In addition, the data stream functionalities in the Qt framework provide easy way to implement interfaces with good interoperability. As, the data stream automatically convert data to the most significant byte (MSB) order format.

7 DISCUSSIONS AND CONCLUSIONS

The objectives for this thesis were set to evaluate the feasibility of Qt framework for advanced communication middleware use and to analyze the advantages and disadvantages for using the Qt. The feasibility study was done by remake existing Peer-to-Peer middleware concept using the Qt framework and the Qt mobility extension APIs. The advantages and disadvantage of the Qt framework were evaluated in the PeerHood quality evaluation. The new PeerHood quality was evaluated with the defined quality factors.

Before Qt based implementation was done, the PeerHood concept and requirements of that was clarified. In addition, the basic knowledge for use of the Qt framework and the Qt Mobility extension was gathered. The Qt framework study was done in communication middleware point of view by leaving all GUI related functions out of scope.

7.1 Qt Framework in Middleware Use

The PeerHood concept was previously implemented on Linux environment to be available in different kind Linux based systems, like Internet tablets or mobile phones. The new PeerHood implementation was decided to remake using the Qt framework. Obviously, the previous PeerHood system was used as a base for the new implementation. The new PeerHood implementation was not targeted mainly on any specific platforms. Purpose was to create the new PeerHood to be cross-platform middleware by using the Qt framework.

The new PeerHood is implemented almost fully with the Qt framework in cross-platform manner. Currently, the only exception is the ICMP ping implementation, which is used only in the WLAN network plugin implementation. All other parts of the new PeerHood implementation are portable between different systems. In the development phase, the PeerHood was tested with the 32 bits and 64 bits Windows 7 and the Ubuntu Linux environments. The portability of the PeerHood implementation is

tested only with these platforms. However, the new PeerHood implementation should be easily portable to the Maemo, Mac OS X and Symbian platforms. Some changes to application deployment might need to be done for each platform.

The Qt based PeerHood implementation was evaluated mainly by comparing it to the existing PeerHood implementation. Especially, resource usage comparisons between these two implementations were very instructive. The differences of CPU and memory usages of the new PeerHood implementation and the old one were significant. Obviously, adding a new abstraction layer will increase the system resource usage. Even though the difference was large, the resource usages for the new PeerHood implementation were in satisfied and acceptable level for use of mobile devices as well.

Probably the biggest advantage of using the Qt framework is portability of it. The Qt framework offers very rich set of fully portable APIs. Usually, communication middleware's are very depended on underlying system. Because of, them uses many platform specific APIs, like sockets, threads and processes. With the Qt framework all these functionalities can be used in cross-platform way.

Second advantage of using the Qt framework is provided event mechanisms. Notably, the signals and slots mechanism provides powerful event mechanism. Using the signals and slots, components can be more loosely coupled and provide almost transparent event delivering from object to another. Hence, the signals and slots mechanisms enable more flexible system design. For event-driven systems – like the PeerHood is – use of the Qt framework does not constraint to any particular system architecture.

As a conclusion of the Qt framework, it provides very sophisticated functionalities for the communication middleware use. With the Qt framework, a lot of middleware functionalities can be implemented to be portable cross-platform components. Even though, the small disadvantage of the system resource usage, the Qt provide mature and efficient environment for flexible cross-platform application and middleware development.

7.2 Future Work

The PeerHood concept is very interesting way to share services between devices in network neighborhood. With the Qt based implementation a new use cases for the PeerHood can be implemented more easily. The good portability of the PeerHood enables use of PeerHood in many platforms including mini laptops, mobile devices, smart phones and desktop computers. The large scale of available systems can provide totally different type systems and better interoperability between mobile and fixed devices.

The implemented Qt based PeerHood system is only a small function set for to prove feasibility of the Qt framework of middleware programming use. In the future a new network plugins can be implemented to provide more extended network use. The PeerHood1 implementation contains network plugins for the GPRS and Bluetooth networks in addition of WLAN and localhost plugins. At least, these two network plugins could be implemented to the new PeerHood for enabling more advanced communication in network neighborhood.

The Qt Mobility extension contains API for the service framework. The Qt service framework has a similar aspect than the PeerHood system by sharing services. PeerHood itself is easy to modify so that PeerHood service is available from service discovery. More interesting study could be integrating the PeerHood system into service framework, so that each local and remote service can be found through of the Qt service framework API. In addition, service publishing to remote devices would be done with service framework. Basically, in integration the PeerHood API is replaced with the Qt service framework and the PeerHood daemon provide backend extension for the service framework.

Another usable addition to the PeerHood could be the Qt meta-object language (QML) [58] support in the PeerHood API. The QML support is for direct use of PeerHood API from the QML context. The QML and Qt Quick [58] enable rapid application UI development for the Qt applications. The QML was introduced in the 4.6 Qt version and the Qt Quick is provided in the Qt 4.7 version.

REFERENCES

1. James Gosling, Bill Joy, Guy Steele and Gilad Brancha. The Java™ Language Specification. Second edition. Prentice Hall. 2005. 688 pages. ISBN 978-0-321-24678-3.

2. Jeffrey Richter. CLR via C#. Second edition. Microsoft Press. 2006. 693 pages. ISBN 978-0-7356-2163-3.

3. Nokia. Qt Modular Class Library [Internet page]. [referred 4.9.2010]. Available: http://qt.nokia.com/products/library

4. Nokia. Supported platforms for the Qt framework [Internet page]. [referred 4.9.2010]. Available: http://doc.trolltech.com/4.6/supported-platforms.html

5. Nokia. Qt Mobility [Internet document]. [referred 5.9.2010]. Available: http://qt.nokia.com/files/pdf/qt-mobility-whitepaper-1.0.0

6. Nokia. The Qt framework licensing options [Internet page]. [referred 5.9.2010]. Available: http://qt.nokia.com/products/licensing

7. Free Software Foundation. The GNU General Public License and the GNU Lesser General public license [Internet page]. Updated 27.4.2010. [referred 4.9.2010] Available: http://www.gnu.org/licenses/

8. Wu Ming-Wei and Lin Ying-Dar. Open Source Software Development: An Overview. Computer. vol. 46, issue 6. pages 33-38. ISSN 0018-9162.

9. Jari Porras, Petri Hiirsalmi, Ari Valtaoja. Peer-to-peer Communication Approach for a Mobile Environment. 37th IEEE Annual Hawaii International Conference on System Sciences. 2004. ISBN- 0-7695-2056-1

10. Stephen H. Kan. Metrics and Models in Software Quality Engineering. 1st Edition. Addison-Wesley Publishing Company. 1994. 344 pages. ISBN 0-201-63339-6.

11. B. W. Boehm, J. R. Brown, M. Lipow. Quantitative evaluation of software quality. Proceedings of the 2nd international conference on Software engineering. 1976.

12. Roger S. Pressman. Software Engineering: a Practitioner's Approach. 5th Edition. McGraw-Hill. 2001. 860 pages. ISBN 0-07-365578-3.

13. Steven Clarke. Measuring API Usability. Dr. Dobbs Journal. May 2004. Pages S6-S9.

14. Erich Gamma, Richard Helm, Ralph Jonson, John Vlissides, Design patterns – Elements of Reusable Object-Oriented software. Addison-Wesley. 1995. 416 pages. ISBN 0-201-63361-2.

15. Jie-Cherng Chen and Sun-Jen Huang. An empirical analysis of the impact of software development problem factors on software maintainability. Journal of Systems and Software. Volume 82, Issue 6. Pages 981-992. Jun 2009.

16. Indar Sugiarto. Static Code Analysis for Software Quality Improvement: A Case Study in BCI Framework Development. Jurnal Informatika. January 2008. Vol 9. No 2.

17. Linda Rosenberg, Ted Hammer, and Jack Shaw. Software metrics and reliability. Technical report, NASA Software Assurance Technology Center. November, 1998.

18. Jerry Gao and Ming-Chih Shih. A Component Testability Model for Verification and Measurement. Computer Software and Applications Conference, 2005. COMPSAC 2005. 29th Annual International. pages 211-218. July 26-28. 2005. ISSN: 0730-3157.

19. Andrew T. Cambell, Geoff Coulson and Michael E. Kounavis. Managing complexity: Middleware explained. IT Professional. IEEE Computer Society. pages 22–28. September/October 1999.

20. Rüdiger Schollmeier. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications. In Proceedings of the IEEE 2001 International Conference on Peer-to-Peer Computing (P2P2001). Linkping, Sweden. August 27-29. 2001.

21. Maemo community. Maemo Software Platform [Internet page]. [referred 23.3.2010]. Available: http://maemo.org/intro/platform/.

22. PeerHood project. PeerHood subsystem specification, version 0.2 [development resource]. [referred 5.9.2010, SVN revision 186]. Available: https://www2.it.lut.fi/svn/public/peerhood/trunk/PeerHood_documentation/specification.doc.

23. PeerHood project. Functional and non-functional PeerHood requirements [Internet page]. [referred 5.9.2010]. Available: http://www2.it.lut.fi/wiki/doku.php/peerhood/specification.

24. Arto Hämäläinen, Jari Porras and Pekka Jäppinen. Service Discovery in Mobile Peer-to-Peer Environment. 5th Workshop on Applications of Wireless Communications (WAWC'07), 2007

25. Nokia. 4.6 Qt Reference Documentation [Internet page]. [referred 25.9.2010] Available: http://doc.qt.nokia.com/4.6/index.html.

26. KDE Community. A Brief History of KDE Project [Internet page]. [referred 23.3.2010]. Available: http://www.kde.org/community/history/.

27. KDE Community. A New KDE Project Announcement [Internet page]. [referred 23.3.2010]. Available: http://www.kde.org/announcements/announcement.php

28. Johan Telin. Foundations of Qt Development. Apress. 2007. 528 pages. ISBN 978-1-59059-831-3.

29. Nokia. Introduction to Qt object model [Internet page]. [referred 20.8.2010]. Available: http://doc.qt.nokia.com/4.6/object.html.

30. Nokia. Introduction to Qt Meta-object system [Internet page]. [referred 20.8.2010]. Available: http://doc.qt.nokia.com/4.6/metaobjects.html.

31. Nokia. Introduction to Qt signals and slots system [Internet page]. [referred 21.8.2010]. Available: http://doc.qt.nokia.com/4.6/signalsandslots.html.

32. Nokia. Introduction to Qt events [Internet page]. [referred 21.8.2010]. Available: http://doc.qt.nokia.com/4.6/eventsandfilters.html.

33. Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann. Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects. Volume 2. John Wiley & Sons. 2000. 633 pages. ISBN 0471606952.

34. The Unicode consortium. Unicode 4 specification [Internet document]. [referred 23.8.2010]. Available: http://www.unicode.org/versions/Unicode4.0.0/.

35. Nokia. Qt containers explained [Internet page]. [referred 23.8.2010]. Available: http://doc.qt.nokia.com/4.6/containers.html.

36. Freedesktop.org project. Introduction to D-BUS [Internet page]. [referred 26.8.2010]. Available: dbus.freedesktop.org.

37. Nokia. QTest library manual [Internet page]. [referred 24.8.2010]. Available: http://doc.qt.nokia.com/4.6/qtestlib-manual.html

38. Nokia. qmake manual [Internet page]. [referred 25.8.2010]. Available: http://doc.qt.nokia.com/4.6/qmake-manual.html

39. Nokia. Qt Mobility API documentation [Internet page]. [referred 9.9.2010]. Available: http://doc.qt.nokia.com/qtmobility-1.0/.

40. Nokia. Qt Mobility API platform compatibility table [Internet page]. [referred 9.9.2010]. Available: http://doc.qt.nokia.com/qtmobility-1.0/#platform-compatibility.

41. Matthias Ettrich. Designing Qt-Style C++ APIs [Internet document]. Trolltech. Qt Quarterly. 2005. [referred 3.9.2010]. Available: http://doc.trolltech.com/qq/qq13-apis.html.

42. Qt Gitorious. Qt Coding Conventions [Internet page]. Updated 25.2.2010. [referred 3.9.2010]. Available: http://qt.gitorious.org/qt/pages/QtCodingStyle.

43. Jussi Laakkonen. PeerHood as UMSIC middleware module. Master's Thesis. Lappeenranta University of Technology. 2009.

44. Herb Sutter. Pimples--Beauty Marks You Can Depend On. C++ Report, from More C++ Gems. Cambridge University Press. 2000. ISBN 978-0521786188.

45. PeerHood project. PeerHood1 implementation [version control system]. [referred 30.8.2010, SVN revision 170]. Available: https://www2.it.lut.fi/svn/public/peerhood/trunk/PeerHood_core.

46. PeerHood2 implementation [version control system]. [referred 30.8.2010, Git tree sha1 9723046]. Available: http://www.gitorious.org/peerhood/peerhood2.

47. Campwood Software. SourceMonitor, a static code analyzing tool [Internet page]. [referred 16.9.2010]. Available: http://www.campwoodsw.com/sourcemonitor.html.

48. Steve McConnell. Code Complete. Second Edition. Microsoft Press. 2004. 960 pages. ISBN: 0735619670.

49. Balister Philip, Dietrich Carl and Reed, Jeffrey H. Memory Usage of Software Communication Architecture Waveform. SDR Forum Technical Conference. 2007.

50. The Valgrind Developers. Massif tool in the Valgrind [Internet page]. [referred 18.9.2010]. Available: http://valgrind.org/docs/manual/ms-manual.html

51. John Nordby. MassifG – a Massif output visualizing tool [Internet page]. Updated 8.2.2010. [referred 19.9.2010]. Available: http://www.jonnor.com/2010/08/introducing-massifg-0-1/.

52. John Berthels. Exmap – a memory analyzing tool [Internet page]. [referred 19.9.2010]. Available: http://www.berthels.co.uk/exmap/.

53. The GCC team. gcov – a tool for instrument and measuring code coverage [Internet page]. [referred 23.9.2010]. Available: http://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

54. Linux test project. lcov – a tool for generate readable output of the gcov results [Internet page]. Update 16.8.2010. [referred 23.9.2010]. Available: http://ltp.sourceforge.net/coverage/lcov.php.

55. Dimitri van Heesch. Doxygen – a source code documentation tool [Internet page]. [referred 23.9.2010] Available: www.doxygen.org/.

56. J. Postel. Internet Control Message Protocol (ICMP) RFC [Internet document]. Network Working Group. 1981. [referred 17.9.2010]. Available: http://www.ietf.org/rfc/rfc792.txt.

57. Bluetooth Special Interest Group. Bluetooth specification [Internet document]. [referred 18.9.2010]. Available: www.bluetooth.com/English/Technology/Building/Pages/Specifcation.aspx.

58. Nokia. Introduction to Qt Quick for C++ developers [Internet document]. [referred 26.10.2010]. Available: http://qt.nokia.com/files/pdf/qt-quick-for-c-developers

APPENDIX 1. Existing PeerHood Interface

```
/**
 * @memo Definition of the PeerHood interface.
 * @doc Definition of the PeerHood interface. This interface defines the public
 * API of the whole PeerHood library.
 *
 */
class MPeerHood
{
 public:
  /**
   * @memo Default destructor.
   * @doc Default destructor. Currently this does nothing but is here because
   * base classes without a destructor are <b><i>evil</i></b>.
   *
   * @return none
   */
  virtual ~MPeerHood() {};


  /**
   * @memo Method used to create a new instance of the PeerHood interface.
   * @doc Method used to create a new instance of the PeerHood interface. The
   * only way to create the instance is via this method. When this method is
   * called for the very first time a new instance of the interface is created.
   * Subsequent calls will return a pointer to the existing instance.
   *
   * @param aCallback Pointer to the callback that will receive notifications
   * from the PeerHood library.
   *
   * @return pointer to a PeerHood instance
   */
  static MPeerHood* GetInstance(CBasicCallback* aCallback = NULL);


  /**
   * @memo Initializes the PeerHood instance.
   * @doc Initializes the PeerHood instance. These routines include connecting
   * to the PeerHood daemon and setting up the debug output. In addition, all
   * internal variables are initialized. This method should be called only
   * once.
   *
   * @param aArgc The number of parameters.
   * @param aArgv Array containing the parameters.
   *
   * @return true if the object was initialized succesfully
   */
  virtual bool Init(int aArgc, char** aArgv) = 0;


  /**
   * @memo Gets a list of all nearby devices and their services.
   * @doc Gets a list containing all nearby devices and their services and
   * resources. Note that this function reserves the memory required by the
   * list and it's caller's responsibility to free it. Also note that the
   * returned list contains <i>all</i> devices in range - inluding those
   * without PeerHood capability. If no devices are found then the returned
   * list is empty.
   *
   * @return a list of found devices or NULL if an error occurred
   */
  virtual TDeviceList* GetDeviceListL() = 0;
```

(continue)

APPENDIX 1. (continued)

```
/**
 * @memo Returns a list of devices that offer the asked service.
 * @doc This function builds and returns a list that contains all devices
 * that offer the requested service. Note that the caller must free the
 * memory allocated for the returned list. If no devices are found then the
 * returned list will be empty.
 *
 * @param aServiceName The service that should be looked for.
 *
 * @return a list of devices that offer the requested service or NULL if an
 * error occurred
 */
virtual TDeviceList* GetDeviceListL(const std::string* aServiceName) = 0;


/**
 * @memo Returns all locally registered services.
 * @doc Returns all locally registered services on a list. The memory
 * allocated for the returned list is not freed automatically so the caller
 * must take care of it. If no services are registered then the returned list
 * will be empty.
 *
 * @return a list of locally registered services or NULL in the case of an
 * error
 */
virtual TServiceList* GetLocalServiceListL() = 0;

/**
 * @memo Creates a connection to a local service.
 * @doc Creates a connetion to a local service.
 * Destination address and technology prototype are taken from the parameters. If a
 * connection object is returned then it's caller's responsibility to delete
 * it in a controlled way.
 *
 * @param aService The service to connect to.
 *
 * @return a new connection object or NULL if an error happened
 */
virtual MAbstractConnection* Connect(TServiceIterator& aService) = 0;

/**
 * @memo Creates a connection to a service on another PeerHood capable device.
 * @doc Creates a connetion to a service on another PeerHood capable device.
 * Destination address and technology prototype are taken from the parameters. If a
 * connection object is returned then it's caller's responsibility to delete
 * it in a controlled way.
 *
 * @param aDevice The remote device.
 * @param aServiceName Remote service's name.
 *
 * @return a new connection object or NULL if an error happened
 */
virtual MAbstractConnection* Connect(TDeviceIterator& aDevice,
                                     const std::string aServiceName) = 0;
```

(continue)

APPENDIX 1. (continued)

```
/**
 * @memo Registers a service so that other PeerHood devices can find it.
 * @doc Registers a service so that other PeerHood devices can find and use
 * it. This method contacts the PeerHood daemon that in turns starts to
 * advert the service through its currently running plugins.
 *
 * @param aName The name of the service.
 * @param aAttributes Service's attributes in one string.
 * @param aPort Service's port.
 *
 * @return port number if the service could be registered, otherwise 0
 */
virtual unsigned short RegisterService(const std::string& aName,
                                       const std::string& aAttributes,
                                       const std::string& aPort) = 0;


/**
 * @memo Registers a service so that other PeerHood devices can find it.
 * @doc Registers a service so that other PeerHood devices can find and use
 * it. This method contacts the PeerHood daemon that in turns starts to
 * advert the service through its currently running plugins.
 *
 * @param aName The name of the service.
 * @param aAttributes Service's attributes in one string.
 *
 * @return port number if the service could be registered, otherwise 0
 */
virtual unsigned short RegisterService(const std::string& aName,
                                       const std::string& aAttributes) = 0;


/**
 * @memo  Unregisters a previously registered service.
 * @doc Unregisters a previously registered service. After unregistration
 * other devices are unable to find and call the unregistered service. Note
 * that the unregistration procedure doesn't delete the service object so
 * this should be done by the actual server application.
 *
 * @param aName The name of the service to be unregistered.
 *
 * @return true if the service could be unregistered
 */
virtual bool UnregisterService(const std::string& aName) = 0;


/**
 * @memo  Unregisters a previously registered service.
 * @doc Unregisters a previously registered service. After unregistration
 * other devices are unable to find and call the unregistered service. Note
 * that the unregistration procedure doesn't delete the service object so
 * this should be done by the actual server application.
 *
 * @param aName The name of the service to be unregistered.
 *
 * @return true if the service could be unregistered
 */
virtual bool UnregisterService(const std::string& aName,
                               const std::string& aPort) = 0;
```

(continue)

APPENDIX 1. (continued)

```
/**
   * @memo Sets a device under constant monitoring.
   * @doc Sets a device under constant monitoring. If a change (out of range,
   * back in range) takes place then the registered callback interface is
   * notified. An application must derive from the <code>CBasicCallback</code>
   * class and implement the defined methods in order to receive callback
   * events.
   *
   * @param aDevice The device that should be monitored.
   *
   * @return true if the monitoring could be started
   */
  virtual bool MonitorDevice(TDeviceIterator& aDevice) = 0;


  /**
   * @memo Stops the monitoring of a device.
   * @doc Stops the monitoring of a device. After this function is called the
   * given device is no longer monitored.
   *
   * @param aDevice The target device.
   *
   * @return true if the monitoring could be canceled succesfully
   */
  virtual bool UnmonitorDevice(TDeviceIterator& aDevice) = 0;


  /**
   * @memo Sets a device under constant monitoring using signal-level monitoring.
   * @doc Sets a device under constant monitoring using signal-level monitoring.
   * If a change (out of range,
   * back in range) takes place then the registered callback interface is
   * notified (Not currently used). An application must derive from the
   * <code>CBasicCallback</code> class and implement the defined methods
   * in order to receive callback events.
   *
   * @param aDevice The device that should be monitored.
   *
   * @return true if the monitoring could be started
   */
  virtual bool SignalMonitorDevice(TDeviceIterator& aDevice) = 0;


  /**
   * @memo Stops the signal-level monitoring of a device.
   * @doc Stops the signal-level monitoring of a device. After this function is called
   * the given device is no longer monitored.
   *
   * @param aDevice The target device.
   *
   * @return true if the monitoring could be canceled succesfully
   */
  virtual bool SignalUnmonitorDevice() = 0;


  /**
   * @memo Sets the plugin prefered by the current application (Not used currently).
   * @doc Sets the plugin prefered by the current application. This means that
   * PeerHood will try to use the given plugin whenever possible. This method
   * will override the value read from the configuration file. The prefered
   * plugin can be changed during runtime. However, it affects only the actions
   * performed after the call i.e. the running services are not affected.
   *
   * @param aPluginName The name of the prefered plugin.
   *
   * @return none
   */
  virtual void SetPreferedPlugin(const char* aPluginName) = 0;
};
```

APPENDIX 2. Feature Comparison between PeerHood1 and PeerHood2

Requirement	**PeerHood 1**	**PeerHood 2**
Device discovery	Implemented	Implemented
Service discovery	Implemented	Implemented
Service sharing	Implemented	Implemented
Connection establishment	Implemented	Implemented
Active monitoring of a device	Implemented; signal level and ping based monitoring	Implemented; signal level and ping based monitoring. Ping based implementation is only available on Linux environment
Data transmission between devices	Implemented	Implemented
Seamless connectivity	Implemented	Not implemented; implementation was left out of scope
Network management	Implemented	Implemented
Component management	Implemented	Not implemented; event system implemented for network plugins. The Qt and Qt Mobility provides needed events
Communication concurrency base	Implemented; based on multithreading	Implemented; single thread based on Qt event loop system
Event interface	Implemented**;** based on callback interface	Implemented; based on use of Qt signals
Plugin architecture for networks	Implemented; WLAN, GPRS, Bluetooth and localhost plugins are available	Implemented; WLAN and localhost plugins are available.
User control	Not implemented	Not implemented

APPENDIX 3. New PeerHood API

```
/****************************************************************************
**
**  Copyright (C) 2010 Kimmo Kolehmainen,
**                     kimmo@omamaailma.net,
**                     www.omamaailma.net
**
**  Copyright (C) 2010 Lappeenranta University of Technology,
**                     Information Technology,
**                     Communications software laboratory
**
**  All rights reserved.
**
**  PeerHood is free software: you can redistribute it and/or modify
**  it under the terms of the GNU Lesser General Public License
**  version 2 as published by the Free Software Foundation.
**
**  PeerHood is distributed in the hope that it will be useful,
**  but WITHOUT ANY WARRANTY; without even the implied warranty of
**  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
**  GNU Lesser General Public License for more details.
**
**  You should have received a copy of the GNU Lesser General Public
**  License along with PeerHood. If not, see <http://www.gnu.org/licenses/>.
**
****************************************************************************/

#ifndef PEERHOOD_H
#define PEERHOOD_H

// INCLUDES
#include <QtCore/QObject>
#include <QtCore/QStringList>

#if defined(PHLIBRARY_LIBRARY)
#  define PHLIBRARYSHARED_EXPORT Q_DECL_EXPORT
#else
#  define PHLIBRARYSHARED_EXPORT Q_DECL_IMPORT
#endif

// NAMESPACES
namespace PH {

// FORWARD DECLARATIONS
class Device;
class Service;
class AbstractConnection;
class PeerHoodPrivate;

/**
  * @enum EDeviceStatus
  * Enumeration for device status provided by monitor functionalities
  * @value DeviceLost, event for monitored device found
  * @value DeviceFound, event for monitored device lost
  * @value WeakLink, event for weak link, weak link is notified when signal
  *         strength goes below of 25% of maximum signal strength
  * @value VeryWeakLink, evetn for very weak link, very weak link is notified
  *         when signal strength goes below of 10% of maximum signal strength
  */
enum EDeviceStatus
{
    DeviceLost = 0x00,
    DeviceFound,
    WeakLink,
    VeryWeakLink
};
```

(continue)

APPENDIX 3. (continued)

```
/**
  * @class PeerHood
  * @brief PeerHood class provide user access to PeerHood middleware. With this
  *        class user can get services and devices from network neighbourhood
  *        and initiate connection to services. User can also set devices to
  *        be monitored and get notifications when device is out of range and
  *        back in range. With PeerHood user can register new services to be
  *        published on other devices and local PeerHood applications.
  * @sa Device Service AbstractConnection
  */
class PHLIBRARYSHARED_EXPORT PeerHood : public QObject
{
    Q_OBJECT
public:

    /**
      * instance
      * Singleton implementation,  this is only way to access to peerhood
      * instance
      * @return PeerHood*, singleton peerhood instance, ownerhip of instance
      *         is not moved.
      */
    static PeerHood* instance();

    /**
      * ~PeerHood
      * default virtual c++ destructor
      */
    virtual ~PeerHood();

    /**
      * init
      * init MUST call before peerhood can be used.
      * @return bool, status of initialization. True if peerhood is ready to
      *         use and if false is returned peerhood cannot use. The most
      *         probably in case of false is returned the peerhood daemon is
      *         not running.
      */
    virtual bool init();

    /**
      * deviceList
      * @return const QList<Device*>, list of currently available devices. This
      *         is constantly changing information and next call content of the
      *         list might be totally different. List is a snapshot of current
      *         situation. Ownerships of device instances are moved to caller
      *         and caller is responsible to free instances when they are not
      *         needed.
      */
    virtual const QList<Device*> deviceList();

    /**
      * deviceList
      * Returns filtered device list.
      * @param const QString&, service which wanted to be in the returned
      *        devices
      * @return QList<Device*>, returns filtered device list. All returned
      *         devices will have given service. Ownerships of device instances
      *         are moved to caller and caller is responsible to free instances
      *         when they are not needed.
      */
    virtual const QList<Device*> deviceList(const QString& service);
```

(continue)

APPENDIX 3. (continued)

```
/**
  * localServiceList
  * @return const QList<Service*>, list of currently available services
  *         registered on local device. This is constantly changing
  *         information and next call content of the list might be totally
  *         different. List is a snapshot of current situation. Ownerships
  *         of device instances are moved to caller and caller is
  *         responsible to free instances when they are not needed.
  */
virtual const QList<Service*> localServiceList();

/**
  * registerService
  * Method provide ability to register services to be published to other
  * devices and local PeerHood applications.
  * @param const QString&, name of service
  * @param const QStringList&, attributes for service
  * @param unsigned int, preferred port to be used
  * @return int, used service port
  */
virtual int registerService(const QString& name,
                            const QStringList& attributes = QStringList(),
                            unsigned int port = 0);

/**
  * unregisterService
  * @param const QString&, name of service to be removed. The removed
  *        service must be registered in the same process
  * @param unsigned int, port number of service to be removed. This value is
  *        optional. Status of remove operation, true if service is
  *        removed.
  * @return bool
  */
virtual bool unregisterService(const QString& name, unsigned int port = 0);

/**
  * connectToService
  * Initiate connection to service in given device.
  * @param Device*, device where wanted to be connected
  * @param const QString&, name of service where wanted to connected.
  * @return AbstractConnection*, connected connection instance or null if
  *         cannot connect or service is not available. Ownership of
  *         instance is moved to method caller.
  */
virtual AbstractConnection* connectToService(Device* device,
                                             const QString& servicename);

/**
  * connectToService
  * Connect to service located on local device.
  * @param Service*, service which wanted to be connected. Ownership of
  *        instance is not moved.
  * @return AbstractConnection*, connected connection instance or null if
  *         cannot connect. Ownership of instance is moved to method caller
  */
virtual AbstractConnection* connectToService(Service* service);

/**
  * hasPendingConnection
  * Method for check is pending connections in given port.
  * @param int, (service)port number
  * @return bool, true if pending connection(s) available.
  */
virtual bool hasPendingConnections(int servicePort);
```

(continue)

APPENDIX 3. (continued)

```
    /**
      * nextPendingConnection
      * Method for accepting pending connection for service.
      * @param int, (service)port number.
      * @return AbstractConnection*, Accepted connection if any available.
      *         If there is no connections pendign for that service port
      *         null is returned. Ownership of instance is moved to method
      *         caller.
      */
    virtual AbstractConnection* nextPendingConnection(int servicePort);

    /**
      * monitorDevice
      * Start monitor actively given device.
      * @param Device*, device which wanted to being monitored. Ownership is
      *        not moved.
      * @return bool, true if monitoring was started succesful.
      */
    virtual bool monitorDevice(Device* device);

    /**
      * unMonitorDevice
      * Stop device monitoring.
      * @param Device*, Device which are monitored and wanted to stop
      *        monitoring. Ownership is not moved.
      */
    virtual void unMonitorDevice(Device* device);

    /**
      * signalMonitorDevice
      * Start signal monitoring given device. Connection strenght which are
      * monitored depends a lot of connection type. It can be signal strength
      * between device to device or connection between device to access point.
      * @param Device*, device which connection is wanted to be monitored.
      *        Ownership of instance is not moved.
      * @return bool, true if monitoring was started succesful.
      */
    virtual bool signalMonitorDevice(Device* device);

    /**
      * signalUnMonitorDevice
      * Stop signal monitoring of given device.
      * @param Device*, Device which are monitored and wanted to stop
      *        monitoring. Ownership is not moved
      */
    virtual void signalUnMonitorDevice(Device* device);

signals:
    /**
      * newConnection
      * This signal is emitted when new connection is arrived. PeerHood cannot
      * accept connections before this signal is not connect to any slot.
      * @param int, service port number, where connection is tried to establish
      * @param int, id for connection.
      */
    void newConnection(int servicePort, int connectionId);

    /**
      * deviceStatusChanged
      * This signal is emitted when monitored device connection changed.
      * @param PH::EDeviceStatus, notified event.
      * @param QString, address of devices which status was changed.
      */
    void deviceStatusChanged(PH::EDeviceStatus status, QString address);
```

(continue)

APPENDIX 3. (continued)

```
protected:
    /**
      * Singleton implementation. Prevent other instance creations.
      */
    explicit PeerHood(QObject *parent = 0);

    /**
      * This is for monitoring is newConnection signal connected-
      */
    void connectNotify(const char* signal);
    /**
      * This is for monitoring is newConnection signal disconnected.
      */
    void disconnectNotify(const char* signal);

protected: // data
    // private implementation to protect binary compatibility
    PeerHoodPrivate* d;

private:
    // Disable copy of PeerHood instance.
    Q_DISABLE_COPY(PeerHood);
};

} //namespace PH

#endif // PEERHOOD
```