

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY

Master's Degree Programme in Information Technology

PAIMEN, LAURI: Mobile Networking with Qt

Master of Science Thesis, 44 pages, 3 Appendix pages

January 2010

Keywords: Mobility, Mobile networking, Qt, Bearer management, Roaming

Using of network for data communications with mobile devices has become mainstream. While keeping the basics the same, mobility introduces new challenges and aspects to the networking. The introduced changes translate to new requirements for the application running in the device, and application developers have to know how to handle them gracefully.

This thesis discusses mobile networking in general, and with Qt framework in particular. The goal is to provide a developer insight into the changes introduced by mobile networking, and present a solution the Qt has to offer.

The biggest change due to the mobility is the usage of third-party networks, which introduces some problems the developer has to recognize. Also, with careful design of network usage the power consumption of the mobile device can be decreased, and the operation time lengthened. Methods of conventional networking are not *on par* with new aspects such as roaming, service networks (SNAPs) and multihoming, giving more headache to the developer.

Qt framework rolls out a cross-platform solution to mobile networking with Mobility API extension. Bearer management library inside it provides an uniform way to handle application connectivity by representing it as a manageable session. This approach makes it easy to implement support for features like roaming and service networks. Qt also incorporates automatic HTTP level roaming, which makes it simple to use services provided over HTTP.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO

Tietotekniikan koulutusohjelma

LAURI PAIMEN: Mobile Networking with Qt

Diplomityö, 44 sivua, 3 liitesivua

Tammikuu 2010

Avainsanat: Liikkuvuus, mobiiliverkot, Qt, yhteyksien hallinta, verkkovierailut

Tietoverkon käytöstä mobiililaitteilla on tullut arkipäivää. Vaikka perusasiat pysyvät samoina, tuo liikkuvuus uusia haasteita ja näkökulmia verkon käyttöön. Nämä muutokset aiheuttavat uusia vaatimuksia laitteissa oleville sovelluksille, joten sovelluskehittäjän täytyy tietää kuinka ne hallitaan.

Tämä diplomityö käsittelee liikkuvaa tiedonsiirtoa sekä yleisesti että Qt-sovelluskehityksen näkökulmasta. Tavoitteena on tarjota kehittäjälle näkemys liikkuvuuden tuomista muutoksista ja esitellä Qt:n tarjoama ratkaisu.

Suurin liikkuvuuden tuoma muutos on kolmannen osapuolien verkkojen käyttö, josta johtuvat haasteet kehittäjän tulee tunnistaa. Verkon käytön suunnittelulla voidaan lisäksi vähentää virrankulutusta ja pidentää laitteen toiminta-aikaa. Näiden lisäksi ongelmia tuottavat perinteiset rajapinnat, joiden tarjoama tuki ei ole pysynyt uusien toimintojen, kuten verkkovierailujen tai palveluverkkojen (SNAP), tasalla.

Qt tarjoaa alustariippumattoman ratkaisun liikkuvan tiedonsiirron hallintaan Mobility API -laajennoksella. Sen sisältämä *bearer management API*, yhteyspisteiden hallintarajapinta, tarjoaa yhtenäisen tavan hallita sovelluksen havainnoimaa verkkoa. Rajapinta esittää verkonhallinnan istuntona, jolloin tuki uusille toimintoille on helppo toteuttaa. Sovelluskehitys tukee lisäksi automaattista HTTP-tason verkkovierailua, mikä helpottaa HTTP:n avulla toteutettujen palveluiden käyttöä.

PREFACE

This thesis was carried out during autumn and winter 2009. I would like to thank the whole open source community for providing open and expressive tools, that made this thesis a joy to work with.

This version of the thesis is licensed under a Creative Commons Attribution-Share Alike 1.0 Finland License, and a modifiable version is available from the author for you to enhance.

Tampere, Finland, January 2010

Lauri Paimen

`lauri@paimen.info`

TABLE OF CONTENTS

| | |
|---|----|
| 1. Introduction | 1 |
| 2. Conventional networking | 3 |
| 2.1 Internet protocol suite | 3 |
| 2.1.1 Internet Protocol | 4 |
| 2.1.2 Transmission Control Protocol | 4 |
| 2.1.3 Hypertext Transfer Protocol | 4 |
| 2.2 Common networking services | 5 |
| 2.2.1 POSIX sockets | 6 |
| 2.2.2 Web services | 7 |
| 2.3 Sample networks | 9 |
| 2.3.1 Home network | 9 |
| 2.3.2 Corporate network | 10 |
| 3. Mobile networking | 12 |
| 3.1 Mobile networking characteristics | 12 |
| 3.2 Bearer management | 13 |
| 3.2.1 Discovery | 14 |
| 3.2.2 Selection | 15 |
| 3.2.3 Roaming | 15 |
| 3.2.4 Multihoming | 16 |
| 3.3 Service networking concept | 17 |
| 3.4 Conventional mobile networking | 18 |
| 4. Mobile networking with Qt | 19 |
| 4.1 Background | 19 |
| 4.2 Licensing | 21 |
| 4.3 Technical features of Qt | 22 |
| 4.3.1 Signals and slots | 22 |
| 4.3.2 Meta-Object Compiler, moc | 23 |
| 4.3.3 QObject trees | 23 |
| 4.3.4 QObject properties | 23 |
| 4.3.5 QtNetwork module | 24 |
| 4.4 Mobility API | 25 |
| 4.5 Bearer Management Library | 26 |
| 4.5.1 Network configuration | 27 |
| 4.5.2 Network session | 28 |
| 4.5.3 System capabilities | 29 |
| 4.6 HTTP level roaming | 30 |
| 4.7 An example of system mobility | 31 |

| | |
|--|----|
| 5. Sample application: FTPsync | 34 |
| 5.1 Architecture | 34 |
| 5.2 Bearer management | 36 |
| 5.2.1 Behavior | 36 |
| 5.2.2 Interaction with bearer management library | 37 |
| 5.2.3 Application specific management | 41 |
| 5.3 Conclusions from FTPsync | 42 |
| 6. Conclusion | 44 |
| References | 45 |
| A. <code>NetworkSessionManager</code> implementation | 47 |

ABBREVIATION AND DEFINITIONS

| | |
|-------|--|
| ADSL | Asymmetric Digital Subscriber Line |
| ALR | Application Level Roaming |
| API | Application Programming Interface |
| BSD | Berkeley Software Distribution |
| CoA | Care of Address |
| CORBA | Common Object Request Broker Architecture |
| DHCP | Dynamic Host Configuration Protocol |
| DNS | Domain Name System |
| FTP | File Transfer Protocol |
| GPL | GNU General Public License |
| GPLv3 | GNU General Public License version 3.0 |
| GPRS | General Packet Radio Service |
| GUI | Graphical User Interface |
| HTTP | Hypertext Transfer Protocol |
| HTTPS | Hypertext Transfer Protocol Secure |
| IAP | Internet Access Point |
| ICMP | Internet Control Message Protocol |
| IDE | Integrated Development Environment |
| IETF | Internet Engineering Task Force |
| IP | Internet Protocol |
| KDE | K Desktop Environment |
| LAN | Local Area Network |
| LGPL | Lesser (earlier; Library) General Public License |
| MMS | Multimedia Messaging Service |
| moc | Meta-Object Compiler |
| MOS | Meta-Object System |
| NAT | Network Address Translation |
| OSI | Open System Interconnection |
| PC | Personal Computer |
| POSIX | Portable Open System Interface (for Unix) |
| QoS | Quality of Service |
| QPL | Q Public License |
| REST | REpresentational State Transfer |
| RFC | Request For Comments |
| SOAP | Simple Object Access Protocol |
| SQL | Structured Query Language |

| | |
|------|----------------------------------|
| SSL | Secure Sockets Layer |
| SVG | Scalable Vector Graphics |
| TCP | Transmission Control Protocol |
| TLS | Transport Layer Security |
| UDP | User Datagram Protocol |
| UI | User Interface |
| URL | Uniform Resource Locator |
| VPN | Virtual Private Network |
| W3C | World Wide Web Consortium |
| WLAN | Wireless Local Area Network |
| WSDL | Web Service Description Language |
| X11 | X Window System |
| XML | eXtensible Markup Language |

1. INTRODUCTION

We live in a connected world. We have built networks spanning the Globe to allow us to share information – *networking*. We have devices in our pockets, giving us a chance to do it anywhere we go, anytime we want – *mobility*. And we have applications to help us do things with the information – *Qt*. Given this, it is up to your imagination to combine the information available on networks with mobility provided by devices, into an application that makes the world a better place. Once you have imagined that, this thesis discusses the practical matters you may encounter when following your dream – when implementing your *mobile networking* application *with Qt*.

To cover the matters, there are three main goals for this thesis. The first one is to study and discuss the challenges and new aspects mobility introduces to networking. The second is to go through the Qt framework in general, and examine how mobile networking issues are addressed in the framework. The third goal is to act as a guide for a developer developing mobile networking applications and join the two other goals together into a comprehensible whole.

To be technically precise, the term "networking" is used to refer to the design, behavior and usage of packet-switched computer networks. "Mobility" is about moving between topologically separate networks. And application is assumed to act as an *client* in the network. Mobile server applications are not addressed.

The thesis is structured into distinct chapters where each discusses one topic around the subject. After the introduction, Chapter 2 goes through the basics of networking. A common protocol stack consisting of IP, TCP and HTTP is visited, with an explanation of how they are used in applications. Chapter 2 also provides an insight on how conventional networks are built with two sample networks.

Chapter 3 focuses on mobility. The effects and characteristics introduced by mobility are discussed first, followed by the actions the mobility requires. Multihoming is examined from mobile networking point of view, and a concept of service networking is presented. Lastly, the chapter explains how mobile networking can be implemented with the methods presented in Chapter 2.

Chapter 4 introduces the Qt framework, and its bearer management library. The chapter begins with an overview of Qt, and explains the licensing model, which is essential to Qt. The following technical review briefly visits Qt's major features. Qt's

forthcoming support for mobile device functionality – Mobility API – is introduced in the correspondingly named section. The rest of the chapter focuses on mobile networking, explaining Qt’s model of bearer management. An alternative method of mobile networking at HTTP level is also presented. The chapter ends to an example of mobile networking behavior from bird’s-eye view.

Chapter 5 gives a practical approach to mobile networking with Qt through a simple sample application. The chosen architecture and bearer management is explained with diagrams, and possible further improvements are discussed. The chapter also includes a short review of bearer management and Qt in general. Chapter 6 concludes the thesis and gives a summary of main points of the subject.

2. CONVENTIONAL NETWORKING

Conventional networks are basic computer networks that are used for example in companies, schools and homes. Computers are connected to local networks, which provide services for the computer. The local network is then connected to a global network, such as the Internet or company-wide network, allowing the computer to reach global services.

The following characteristics can be found from conventional networks and networking:

- Hierarchical design and distinction of local and global network, as introduced.
- Static design. It is often assumed that nodes (computers) rarely move to another location (physically or related to network), and thus little attention is paid to mobility. The structure of the network is usually designed based on location.
- The network equipment is owned or rented, and there is a possibility to administer the network. The quality of the network, such as speed and reliability, is somewhat uniform across the network, and can be improved when needed.

2.1 Internet protocol suite

Internet protocol suite specified by IETF in RFC 1122 defines four protocol layers (Table 2.1) for the Internet and other similar networks [14]. For those familiar with the OSI model [5], the internet layer compares to OSI's network layer and the transport layer contains some session characteristics, but OSI layers below and above those are simplified to link and application layer, respectively. The suite is also known as *TCP/IP model* by two of its most important protocols.

Protocols essential to this thesis are shortly discussed in the following sections.

Table 2.1: Layers of the Internet protocol suite.

| | Layer | Protocols |
|---|-------------|---------------------|
| 4 | Application | FTP, HTTP, SSH, ... |
| 3 | Transport | TCP, UDP, ... |
| 2 | Internet | IPv4, IPv6, ... |
| 1 | Link | Ethernet, PPP, ... |

2.1.1 Internet Protocol

Internet Protocol (IP) is commonly used protocol in packet-switched networks. It delivers packets (datagrams from upper layer protocol) from the source to the destination. The delivery is based on routing by packet's destination address. Internet Protocol is connectionless and provides best-effort delivery without guarantee of order, duplication or reception of the transmitted packets [2, pp. 95-113].

The fourth revision of the protocol, IPv4 [13] is currently the most deployed version. It defines a 32-bit address space having about 4.29 billion addresses, which are speculated to run out in the near future (2011 by Geoff Huston's estimate¹). IPv6 [15] has improvements (128-bit address space, among other things) and new features over IPv4. It is implemented in all major operating systems, but not yet widely deployed.

2.1.2 Transmission Control Protocol

Transmission Control Protocol (TCP) provides reliable, stream-oriented connection. TCP chops the datastream into segments and passes them to Internet Protocol for delivery. Because IP is an unreliable protocol, TCP handles the order, possible duplication and retransmission of the segments. Other features of TCP are error detection, flow control mechanism, and congestion control. [2, p. 249]

While the Internet Protocol works with addresses, TCP connects to port. IP address and TCP port form an abstraction of connection endpoint. Thus both, source and destination endpoint, need to have a port for TCP connection. Port numbers for some services are well-known, like 21 for FTP or 22 for SSH. Usually ports below 1024 are used by system services, and allowed only for administrators or root users.

2.1.3 Hypertext Transfer Protocol

Hypertext Transfer Protocol (HTTP) is a simple protocol developed for retrieving documents from network. The common version in use today is HTTP 1.1, specified in RFC 2616 [16].

HTTP depends heavily on client-server model, where the client initiates action (and connection) with HTTP request, and the server replies. There is no state between request-reply pairs, making HTTP a stateless protocol [2, pp. 530-531]. Both request and reply have a header and a possible body. A header is text-based and human-readable, with encoded body attached when needed. An example of

¹Geoff Huston provides an auto-generated daily report of IPv4 address pool exhaustion at <http://ipv4.potaroo.net>.

Table 2.2: Example of HTTP request and reply. ²

Request:

| | | |
|---|-----------------------|---|
| 1 | GET / HTTP/1.1 | GET request for resource / using HTTP version 1.1. |
| 2 | Host: www.example.com | The Internet host of the resource. |
| 3 | | Empty line (carriage return and line feed) ends header of the request. This request does not have a body. |

Reply:

| | | |
|-----|--|--|
| 1 | HTTP/1.1 200 OK | Indicates successful request. |
| 2 | Date: Tue, 08 Sep 2009 16:49:08 GMT | Server's time. |
| 3-6 | ... | (snipped) |
| 7 | Content-Length: 438 | Length of the body. |
| 8 | Connection: close | Connection will be closed after the reply. |
| 9 | Content-Type: text/html; charset=UTF-8 | Body type and encoding. |
| 10 | | Empty line. Body begins. |
| 11 | <HTML> | First line of the body. |
| 12- | ... | (the rest of the body) |

request and reply is presented in Table 2.2. HTTP requires a reliable transport and TCP is extensively used. Port 80 is a well-known TCP port for HTTP.

Hypertext Transfer Protocol Secure (HTTPS) adds security to HTTP protocol by sending requests through encrypted SSL or TLS connection. This way, plaintext HTTP transactions cannot be eavesdropped. HTTPS uses different TCP port, 443.

HTTP standard has eight methods that can be performed to resource in a server; HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS and CONNECT. The methods are further categorized into safe and unsafe. The safe ones should not change the state of the server and can be thus cached. GET is the most common of the methods. It requests the specified resource from the server, thus being a safe method. POST submits data to be processed on the server and may modify resources on the server. Therefore it is an unsafe method and should not be cached. An example of GET is given in Table 2.2.

2.2 Common networking services

From the Internet protocol suite, the transport layer (see Table 2.1) is the lowest layer available for use in applications. Berkeley sockets is *de facto* interface to access and use it. Web services (as defined by W3C [21]) are popular and "trendy" way to

²Lines end with carriage return (0x0D) and line feed (0x0A) characters.

utilize network at application layer, being popular due to ease of use and accessibility – outgoing HTTP is rarely firewalled.

2.2.1 POSIX sockets

POSIX sockets is an abstraction of network connections. It is based on Berkeley sockets developed at the University of California (located in Berkeley). Practical reference definition of the API for C programming language is contained in POSIX standard 1003.1 [4]. Other programming languages have adopted the definition, usually with minimal (or no) changes. This makes socket programming straightforward in any language.

POSIX API in a nutshell

The most important functions of the socket API (`<sys/socket.h>`) are shortly discussed in the following list. For a more comprehensive definition, see POSIX Standard 1003.1, 2004 version [4].

- `int socket (int domain, int type, int protocol)` creates a new socket. The socket is represented as an integer, and can be used the same way as file descriptors.
- `int bind (int socket, const struct sockaddr *address, socklen_t address_len)` binds the *socket* to an *address*.
- `int listen (int socket, int backlog)` marks the *socket* to listen for incoming connections.
- `int accept (int socket, struct sockaddr *address, socklen_t *address_len)` accepts a connection when the *socket* is set to listen for them, therefore creating a server. The *socket* is used only for accepting new connections. A new socket is created and returned for actual data transmission.
- `int connect (int socket, const struct sockaddr *address, socklen_t address_len)` connects the *socket* to an endpoint at *address*.
- `ssize_t recv (int socket, void *buffer, size_t length, int flags)` receives data through the *socket*. The return value indicates the amount of data received, or a negative error code.

- `ssize_t send (int socket, const void *buffer, size_t length, int flags)` sends data through the *socket*. The return value indicates the amount of data sent, or a negative error code.
- `int shutdown (int socket, int how)` shuts down send and/or receive operations of the *socket*. Being a file descriptor, `int close (int fd)` is also commonly used to shut down both operations.

Usage with TCP

A simple connection to a predefined address and port (client-like functionality) can be achieved by creating a new socket with `socket()`, and connecting it to the endpoint with `connect()`. The socket is implicitly bound to an appropriate interface based on the destination address, and connection establishment starts. When `connect()` is successful, the connection is established, and the socket is ready for transmitting (`send()`) and receiving (`recv()`) application-level data. After transmissions are completed, the socket can be shut down (`shutdown()`). [2, pp. 415-423]

A basic server is only a bit more complicated. After the creation of a socket, it needs to be bound (`bind()`) to a local endpoint (an address and a port), and set to listen for connections (`listen()`). After that, `accept()` can be used to wait for connections to the bound endpoint. Data transmission is done to the socket returned by `accept()` (not to the one being listened), and when everything for the client is done, the returned socket can be shut down (`shutdown()`). If no more clients have to be served, for example when the server is shutting down, the listened socket is shut down. [2, pp. 415-425].

The real art of server programming lies in multiple concurrent connection handling, which falls outside the scope of this thesis.

2.2.2 Web services

In addition to the transport layer, applications can use application layer protocols and APIs built on top of them. Often called web services, the APIs offer an application and platform independent way to exchange information [21]. We concentrate on the HTTP stack mainly for the following reasons:

1. Firewalls and proxies generally allow HTTP traffic.
2. Ease of use and popularity, partly because many frameworks support HTTP at least on some level.

The usage of HTTP has exploded with ever-growing number of services. In a way, it has become a transport layer for applications.

Web services using HTTP are not gone through in the same detail as POSIX sockets. Rather, two approaches to build a web service are represented. Those are SOAP and representational state transfer.

SOAP

SOAP (once Simple Object Access Protocol) is a protocol specification by W3C [20], using XML to exchange information. It is not specified to use HTTP for exchange, so SOAP can be used on top of other application level protocols, too.

A common use for SOAP is to offer a certain API to be used over the Internet. While there are other standards – such as CORBA – to accomplish the same, SOAP seems to be popular perhaps due to the possibility to use HTTP, and the simplicity and generality of the XML it utilizes. The API can be described with web service description language (WSDL), helping the client side code generation. Many frameworks have either built-in or add on (like Qt) SOAP support.

Representational state transfer

While SOAP is a protocol, representational state transfer (REST) is an architectural style, born from observation and analysis of the Internet, and HTTP particularly. It presents a fundamental view on how Internet-scale services should be built.

REST is a client-server architecture, where communication is stateless. A client holds the state of the session, and every request it makes is self-descriptive allowing the server to fulfill the request without any per-client context. The lack of communication state improves reliability (no partial actions) and scalability (as long as server can handle the requests, the number of clients does not matter). REST encourages uniform interface between components, decoupling the implementation and allowing independent evolution of the components. The communication is layered so that component does not have to see beyond intermediary communication party. With the uniform interface, communication can be optimized with caches (speed), proxies (redirecting) and firewalls (security). REST also has a notion of code-on-demand, which allows functionality of clients to be extended by downloading and executing server-defined code. [3]

REST pays a lot of attention to data elements. Information is abstracted as a resource and further divided into static and variable resources. Static mapping is required to retain the semantics of the resources over the time, and resource identifiers are used for that. Communication is abstracted into representations, where client's request represents an action to a certain resource, and server's reply is an representation of the action's result. [3]

REST is heavily influenced by HTTP's success, and being compatible with each

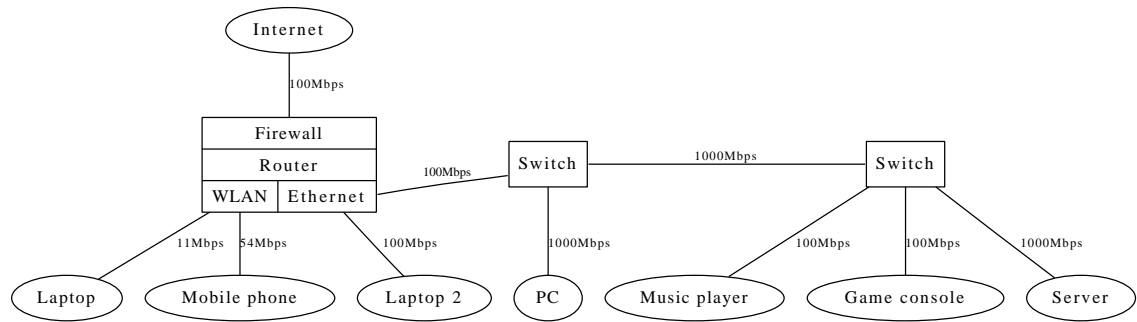


Figure 2.1: Sample home network and link speeds.

other, HTTP is commonly utilized when implementing the architecture.

2.3 Sample networks

Two samples are introduced here to provide a hands-on approach to networking. The first one, home network, is an example of network used for providing a basic connectivity for a handful of devices. The second is a more advanced example, sketching how networks are constructed on a larger scale.

2.3.1 Home network

Figure 2.1 shows a basic home network. This particular network represents author's home network, and is an actual and working setup. Devices are connected either through ethernet or WLAN depending on what is supported by the device and preferred by the user. Some common network components can be found from the setup.

- **Router** is the component usually creating the home network. Nowadays most entry-level routers have an ethernet *switch* and WLAN, which form a local area network (LAN). Private IP address space (usually $192.168.x.0/24$) is reserved for the local network. A gateway to the Internet is provided with NAT³, and a basic *firewall* setup is also possible, making the network secure against certain kind of attacks from the Internet. The router can be integrated to an ADSL modem, thus minimizing the amount of devices and easing the network setup. In Figure 2.1, there are two additional switches used to extend the wired network.
- **Services** can be offered by any device connected to the LAN. The router usually offers some basic networking services, like DHCP and wireless authentication, which are needed to connect to the network. Advanced services, like

³Network Address Translation is a common technique to form a LAN behind one public IP address.

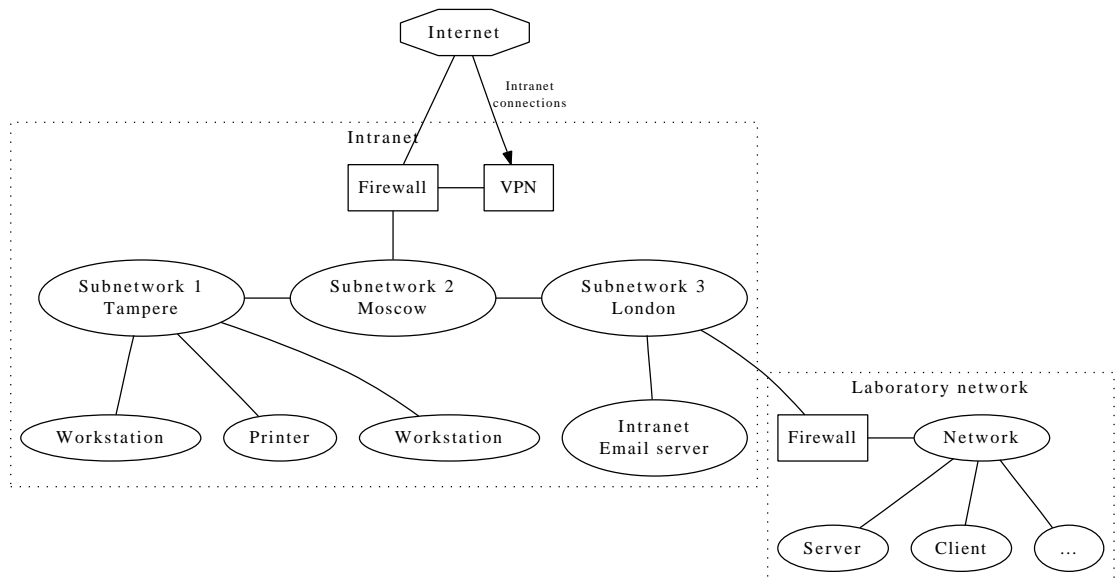


Figure 2.2: Sample corporate network.

shared storage or printing services, can be offered by locally connected devices. The services are only available on LAN due to firewall features of the router, and additional configuration is needed to the router if the service (such as secure shell or web server) has to be accessible from the Internet. In the example network, the server offers its storage to local devices through FTP/NFS/Samba⁴ services. Thus documents are always in one place and can be easily shared and backed up.

- **Clients** – computers and other devices utilizing the network and its services – are usually connected to the router. Often they just go to the Internet, but sometimes they also use the services offered locally. One example in Figure 2.1 is the music player, which can play music stored on the local server or radio streams from the Internet. There is no distinction between server and client. Thus the client can also have a role of the server, and provide services to the network.

2.3.2 Corporate network

A corporate network builds on the same components as the home network – firewalls, routers, switches, servers and clients. It differs in size and is pictured on a logical level rather than component level. Due to bigger size, corporate network is often divided into subnetworks based on location (department), purpose (laboratory network), or other criteria. Services can be divided to three categories, global (Internet), company-wide (intranet) and local services (like printing). Unlike for

⁴Different storage services are needed due to different client support.

the home network, connectivity to the Internet from corporate network may not be necessary or even allowed due to security, performance, or other reasons.

Corporate networks often allow access from outside to some extent, for example to employees working from home or coffee shops. Virtual private networks (VPN) are extensively used for this, allowing mobile clients to connect to the intranet and use the services it provides.

Figure 2.2 represents connectivity of a simple, imaginary corporate network. The network has two sections; the intranet and the laboratory network. Intranet is built on three location-based subnetworks and has the possibility for *nomad employees*, such as travelling salesmen, to connect to the company's intranet through the VPN. Laboratory network is connected to one of the subnetworks, and thus it is accessible from the whole intranet. The firewall between subnetworks keeps the networks apart; if the laboratory network happens to be in a laboratory observing computer viruses, it keeps the viruses from spreading to the intranet (and in worst case, to the Internet).

3. MOBILE NETWORKING

Mobile networking is often discussed as inter- and intra-access, or respectively, *micro-* and *macro-*mobility. Micro-mobility is mobility *inside* a network, such as inside a 3G network, where handover between network access points can be controlled. Thus, an illusion of continuous network can be created in link layer, hiding some problems of mobility. Macro-mobility is mobility *between* networks, such as 3G and WLAN, where the networks are not "compatible" – *i.e.* different network technologies, no common administration, and so on. In this situation, the change of the network usually breaks IP level connectivity. Many protocols have been proposed to address this and other issues of macro-mobility.

This chapter discusses macro-mobility from the perspective of mobile client, with a notion of service networks.

3.1 Mobile networking characteristics

Mobile networking is basically just a combination of *using a network*, and *moving around*. *Using a network* is a problem partially solved in the previous chapter. The network can be used the same way as conventional networks – with sockets or higher level APIs. However, this does not fully solve the problem, because *moving around* exposes us to a new field of problems. What should happen when one moves out of reach of a network? Or to reach of another network? What if the usage of the old network costs money and the new is free, but the change would discard all established connections? These are some examples associated with the field we are discussing.

Mobility emphasizes some characteristics that have been less essential in conventional networking. Instead of stability, the environment is constantly changing. This means constant reconnecting, freezes in traffic, and partial replies. While it is good even for a conventional application to be prepared for these, in mobile networking they are definitely an issue. It may require a notable amount of work to provide a good user experience when the network is "constantly" failing.

Outside the reach of "own" network, mobile devices can use third-party networks, such as WLANs or 3G, to provide connectivity. The main difference between own and third-party network is the administrator and control; a third-party network has its own management. Thus the use of third-party networks poses some characteris-

tics and challenges.

- Usage costs. Connectivity costs may be based on connection time (modems), transferred data (3G networks), periodic (monthly) fees, pay-per-use (internet cafés), location (long-distance calls, 3G abroad), or mix of basically anything. The variety of costs makes it hard to estimate the total expense and complicates the decision whether to use the network.
- Varying connectivity. It might be that the service is filtered (used port is firewalled) or unreachable (not connected to the Internet) in a third party network, thus being unusable even though the device itself is connected. Also, network throughput and latency may vary dramatically. If a laptop "drops" from ethernet to GPRS, the throughput may fall from 1Gbit/s to 28.8kbit/s, and latency increase from tens to hundreds of milliseconds. A third-party network may also require authentication with different mechanisms like WLAN passwords or web-based authentication before connections can be established.
- Lack of advanced functionality. All third-party networks cannot be assumed to support certain functionality, if it requires some "special" components to exist on the network. The lack of components assisting roaming is the most notable functionality in the context of this thesis. Conventional networking functionality with some limitations (mostly by firewalls) can usually be expected from a third-party network.
- Weakened IP level security. Some networks, such as unencrypted WLAN or ethernet hub, may allow anyone to listen to the traffic and extract sensitive information from it. This can be prevented by using only trusted networks (not any available) and transport layer security protocols, such as SSL.

Battery consumption is also an important factor in mobile devices. Networking may draw notable amount of power and thus reduce the operation time of the device considerably. The more connectivity is provided, the more power is consumed. In mobile environments, this has to be taken into account. Basic improvements are to turn unnecessary network interfaces off, reduce the interval of network discovery and adjust the power used for transmitting and receiving data. These may produce minor side effects for applications, like increased connection establishment times. For applications targeting mobile environment, the battery usage of the application should be investigated. By simply closing connections when they are not needed for a while, power can be saved and operation time lengthened.

3.2 Bearer management

Bearer management is the process of using the right bearer for the current purpose.

There are three parties who can choose the bearer, and thus three perspectives to bearer management:

- **User** is usually interested of costs of using the bearer, whether it is monetary or power consumption. Being a human, simplicity and "easyness" of bearer management also counts, which often leads manufacturers to include some bearer management automation into systems. The chosen bearer and management strategy affect the experience user receives.
- **Application** is concerned about reachability and user experience. It may not want to use certain bearer, if the desired destination is not reachable through it. As a part of the user experience of mobile device, the application may also be interested of bearer selection – especially if user's action is needed when connections break.
- **System** handles bearers and routes the traffic. Handling bearers include settings (such as WLAN password) handling and resource management (such as transmission power management). System routes the traffic to the right bearer.

Usually the process of choosing the bearer is divided between the parties, which balances the perspectives. For example, it may be possible for a user to prioritize bearers, which the system then automatically tries in order from the most to the least preferred.

The management of bearers can be divided into three phases which are discussed next. A symptom of multi-bearer management – called multi-homing – is also visited.

3.2.1 Discovery

The result of a discovery is a list of available access points or networks. The more mobile the device is, the more the list "lives" over time.

Discovery may draw some power, such as a scan for available WLAN networks does, and is thus optimized: the discovery is usually done only when actually needed – like when some application has requested network connection – and the results are cached for some time.

Discovery is usually delegated to a system, which does it automatically for application or user. If the discovery draws power, the user is often presented with an option to turn off the automatic discovery, and to perform it manually when necessary.

3.2.2 Selection

After a list of available access points is obtained, the next phase is to select the most appropriate one for use. Conventionally (for example in early laptops, having WLAN and ethernet) the user has affected a lot in this phase, even to the extent of manually selecting the access point each time. Mobile devices, such as phones, have automated the selection in simple cases, and have user-adjustable settings and rules for harder ones, making the selection straightforward in most cases.

Some mobile platforms allow applications to control the selection (for example, restrict only to WLAN), or may consult applications for their opinion of access points.

After the selection is done, application is ready to start networking.

3.2.3 Roaming

Discovery can be continued after the access point is selected. New, more preferable access points can be found, or the currently selected one may wither (QoS drops) or disappear (WLAN goes out of reach). It is time to do some roaming – to decide whether to change to the newly found access point on-the-fly, or to stick with the old one. The term "roaming" usually refers to the first option.

The roaming on micro-mobility (L2, link layer) scale is rather unnoticeable to the application. Therefore we concentrate on macro-mobility (L3, Internet/transport layer) roaming.

The main motivation to continue with the same access point is the achieved connection state. For example, if a heavy login procedure is completed, or unresumable download is going on, it might be impossible to change and start all over again. Another reason for sticking may be that the service is not simply reachable from the new access point, or the new access point does not offer any better service. Characteristics driving to change the current access point are the opposites; the new access point is better in some way (cheaper, faster, ...), the connections are light-weight and renegotiable, or the current one is unusable (out of reach).

Seamless mobility

Roaming to a new access point tends to break the existing connections, which is an undesirable effect. Different protocols have been invented to overcome this limitation. When these protocols can be used, the barrier for roaming is lower because the connections do not break. We shortly review one of these, Mobile IP, which is an IETF standard protocol for maintaining a permanent IP address while roaming between networks.

Mobile IP, specified in RFC 3344 [17] for IPv4, defines two addresses for a mobile

device. A home address is permanent, and care of address (CoA) temporary. When the device roams, it keeps its home agent aware of its current CoA. The home agent receives data from home address and tunnels it to the device. Transmission is done from CoA to destination, but using the permanent home address and foreign agent, which acts as middle man.

Like the Mobile IP, the protocols providing seamless IP-level networking often utilize additional networking components. These components may not be available in all networks, and thus seamless mobile networking can not always be provided.

Automatic and application level roaming

From an application point of view, there is two kinds of roaming. Automatic (sometimes called forced) roaming means that the system (or the user) makes the roaming decisions for application. This usually leads to uniform roaming behaviour on system level and is less laborous to the applications, because there is no need to implement decision logic. On other hand, the roaming may happen anytime to any access point, breaking the connection at a critical moment and leading to unfavourable bearer for the application. On some systems, there is a way to inform the application of roaming, but on other, the application only sees old connections failing (timeouts, resets, etc).

Application level roaming (ALR) is a more sophisticated mechanism, where the system consults applications on the roaming decisions. This allows applications to match the roaming behaviour with application-specific protocols and needs, and generally control the roaming experience. For example, an application can disable roaming while downloading a large, unresumable file from network (yet it might pay off to restart with faster network – it is up for the developer to decide).

Automatic and application level roaming are not exclusive. Both system and application may support the both kinds at the same time.

3.2.4 Multihoming

Multihoming means that instead of (n)one, a device has multiple IP addresses. This situation may arise when multiple interfaces, such as access points and links, are connected simultaneously. Alternatively, a single interface may have multiple IPs. Conventionally multihoming has been used to increase the reliability of links by multiplying them. Routing protocol is then used to recover from a failing link so that the traffic flow continues as normally as possible.

In mobile networking, multihoming still means that multiple networks are connected simultaneously, but the reason and outcome is different. One reason is roaming. For example, when application A roams to a newly found access point and

application B does not, it leads to multihoming. In this case, the device needs to maintain a connection to the current access point, and open a new one to the found access point. Another reason – in mobile phone industry – is that MMS messages must be sent through a certain 3G network, which is not suitable for browser, for example. Thus, if a MMS is sent while browsing, both networks must be connected. This makes the device multihomed.

Multihoming poses some challenges to the device. First of all, better connectivity means bigger power consumption, and possible extra costs if done carelessly. Secondly, there is a routing problem when two networks have overlapping IP spaces. If link A and B both provide access to 192.168.0.0/24¹, through which link a socket to 192.168.0.40 should be connected? Links may be connected to different networks and with a wrong choice, the endpoint is unreachable. Solution for this is presented later in Qt bearer management (Section 4.5). Finally, connections may render others "impossible". For example, usually only one WLAN access point may be connected, or incoming call ceases data traffic in 2G network. The device needs to take these into consideration.

3.3 Service networking concept

A service network is simply a group of access points representing a network offering certain services. A practical example is a corporate network, which can be accessed through VPN from the Internet, and WLAN or ethernet at the office. The essential thing is that the corporate network provides services, which are available through certain mechanisms (like the VPN) or access points (like the WLAN or ethernet). Thus, by grouping these access points to a service network, the system can provide applications a simple way to connect to these services – just connect to the service network.

Basically, "service networks" could be offered as an access point (but then we would not need the term). VPN is an example of such case, where by simply connecting to VPN, the services are available for application. A single access point is sufficient also for MMS, because the MMS messages are sent over cellular network. However, if the services can be accessed through multiple access points, mapping services to certain access point is certainly not a suitable solution. Also, by grouping the access points, applications are allowed to roam "inside" the service network. One network available from multiple access points is the Internet, which can also be understood as a service network. Usually the Internet is default service network for the applications that do not implement service network support.

¹192.168.0.0/24 is a common IP address space for private network. The space spans from 192.168.0.0 to 192.168.0.255.

3.4 Conventional mobile networking

This section discusses various conventional methods that can be used with mobile networking, if there is no framework support available.

On socket (transport) level, sockets can be bound to a specific address. By binding a socket to the IP address of a desired interface, the socket can be set to use that interface. When the socket is connected (or accepted), traffic flows through the desired interface.

An alternative way to bind a socket to a certain interface is to use `SO_BINDTODEVICE` socket option, which bypasses the routing table and binds the socket straight to the device or interface. Unfortunately, only Linux has support for this option. For the case where the destination network has a unique IP address block, socket option `SO_DONTROUTE` may be an appropriate way to connect to that network when it is available. `SO_DONTROUTE` sets the socket to use the interface which is connected to the destination directly without routing (IP address and subnet mask match).

However, IP addresses tend to change constantly while roaming, making the network selection a tedious job. Every socket has to be recreated, rebound and reconnected. It should also be noted that interface may not be connected to the same network all the time, which is the case for WLAN interface, making binding to the certain device useless in some cases. If multiple interfaces can connect to multiple networks, figuring out the right interface for the desired network is even harder.

VPN is another tool for mobile networking. It is commonly used to reach a certain service network, as in the corporate network example (Section 2.3.2).

Mobility Support for upcoming IPv6 is proposed in RFC 3775 [18]. Mobility Support is similar to Mobile IP (for IPv4), and improves seamless connectivity when supported. One of the main differences to IPv4 implementation is the lack of "foreign agents", which means that no additional features are needed from third-party network, making it easier to implement the Mobility Support. Home agents are still required.

As demonstrated, the methods of conventional networking can be used for mobile networking, but they fall short when sophisticated handling of connectivity is needed. Qt framework continues from here by providing support for service networks and roaming (for applicable platforms).

4. MOBILE NETWORKING WITH QT

Qt is a powerful C++ cross-platform application and UI framework. The framework is divided into modules, and each module provides a support for certain functionality. Table 4.1 provides a general view of Qt's modules and functionality set. Applications written with Qt can be deployed across desktop, mobile and embedded systems without rewriting the source code. Qt includes a cross-platform class library, integrated development tools and a cross-platform IDE. [12]

The beginning of this chapter gives an introduction to Qt, and for those already familiar with it, Section 4.5 continues with the bearer management library.

4.1 Background

The Development of the Qt framework was started in 1990 by Haavard Nord and Eirik Chambe-Eng. They initially needed "an object-oriented display system" and by 1993, the first graphics kernel and widgets were running. Haavard suggested going into business, and in 1995 the first public release (Qt 0.90) was published by Trolltech. By the end of 1996, Qt 1.1 was released with 18 licenses sold to 8 customers. [1]

Around 1996 KDE project was born to create a consistent and easy-to-use desktop environment for Unix. The father of the project, Matthias Ettrich, chose to use Qt toolkit for the project and other programmers joined. Qt became de facto standard for C++ GUI development on Linux. Ettrich later joined Trolltech. [1]

Qt 2.0 was released in 1999 with a new open source license, Q Public License (QPL). Qt has been dual-licensed with a commercial and a free software license since version 0.90, and apparently the latter was now replaced with the QPL. The licensing changed once more in 2000 to commercial and GPL license, when a lightweight X11 replacement, Qt/Embedded (later Qtopia, then Qt Extended, now discontinued) was released. [1]

Qt 3.0, a major redesign of Qt 2.0, was released in 2001. It was supported on Windows, Mac OS X, Unix, and Linux (desktop and embedded). The next major advance, Qt 4.0 was released on 2005 including new containers, advanced model/view functionality, improved 2D framework, and Unicode text editing classes. [1]

On 2008 Nokia acquired Trolltech, which was renamed as Qt Software. On March

Table 4.1: Modules of Qt 4.5

Modules for general development:

| | |
|---------------|--|
| QtCore | Base classes for Qt and other modules. |
| QtGui | GUI classes. |
| QtNetwork | Networking classes (Section 4.3.5). |
| QtOpenGL | OpenGL (3D) support. |
| QtScript | Classes aiding scripting. |
| QtScriptTools | Scripting tools, like debugger. |
| QtSql | Integration to SQL databases. |
| QtSvg | SVG support. |
| QtWebKit | Web browser engine integration. |
| QtXml | Classes for handling XML. |
| QtXmlPatterns | XQuery and XPath support. |
| Phonon | Multimedia framework. |
| Qt3Support | Classes to ease transition from Qt3. |

Qt tool modules:

| | |
|-------------|---|
| QtDesigner | Access to Qt Designer. |
| QtUiTools | Classes for handling Qt Designer forms. |
| QtHelp | Help system. |
| QtAssistant | Qt Assistant (online help) launcher. |
| QtTest | Unit testing classes. |

Extension modules to utilize ActiveX: (Windows only)

| | |
|--------------|----------------------------------|
| QAxContainer | Extension for accessing ActiveX. |
| QAxServer | Qt application as COM server. |

Extension modules to utilize D-Bus: (Unix only)

| | |
|--------|-----------------------|
| QtDBus | Integration to D-Bus. |
|--------|-----------------------|

Table 4.2: Qt availability for different platforms. [12]

| Platform | Notes |
|--------------------|--|
| AIX | For PowerPC. |
| Embedded Linux | For ARM and Intel 32-bit. Emddeded is for systems without X11. |
| HPUX | For PA/RISC and Intel Itanium. |
| Linux | For Intel 32/64-bit. |
| Mac OS X | For PowerPC and Intel 32/64-bit. |
| Solaris | For SPARC and Intel 32-bit. |
| Windows XP & Vista | For Intel 32/64-bit. |
| Windows CE | For Intel 32-bit, ARmv4i and MIPS. |
| S60 | Technology preview. |
| Maemo | Support announced by Nokia. |
| FreeBSD | Community support. |
| NetBSD | Community support. |
| OpenBSD | Community support. |

2009, Qt 4.5 was released and LGPL version 2.1 licensing option was added [9].

Qt is available for various platforms presented in Table 4.2. It is also adopted by many different organizations and companies for various uses. *Nokia*, world leader in mobility and owner of the Qt Software, is porting Qt to S60 and Maemo platforms [8]. *K Desktop Environment* is an open source project and part of Qt's success. KDE is shipped with Qt, and uses heavily its features to produce a high-quality desktop environment [7].

4.2 Licensing

Qt is available in three licensing options [11], which are designed to support various kinds of software development. Thus the development can be open and/or commercial, under GPL or other license, or even closed source.

- Qt GNU GPL v. 3.0 is a free license, available for the development of open source software governed by the GNU GPL version 3.0 (GPLv3). Essentially, the source code must be provided by the software distributor.
- Qt GNU LGPL v. 2.1 (since Qt version 4.5) is free and requires changes made to Qt to be available as source code. Proprietary applications can be created in accordance with the GNU LGPL v. 2.1 (LGPL) terms. A common interpretation of the LGPL is that it is allowed to link an application with a LGPL library as long as its possible for the user to relink with a different version of the library. This license is also suitable for non-GPL open source projects, and also allows converting license to GPLv3 later.

- Qt Commercial Version. The commercial version allows software distribution without source code. It is targeted for developers who do not want to share the source code. This applies to both application and changes made to Qt. GPL or LGPL licensed Qt components can not be incorporated, and Qt Commercial License must be purchased before the development starts.

KDE Free Qt Foundation was formed by Trolltech and KDE e.V.¹ in 1998 to guarantee the availability of the Qt framework for the development of open source software. An agreement between the foundation and Trolltech was made, giving the foundation the right to release Qt under a BSD-style license in case Trolltech discontinues the development of the Qt [19], [6]. BSD licenses are very permissive free software licenses, allowing unlimited redistribution for any purpose. The agreement practically means that Qt will always be available as open source.

4.3 Technical features of Qt

The Qt framework incorporates some unique technical features "under the hood". This section briefly visits some central mechanisms of Qt 4.5. Qt Mobility API (Section 4.4) and classes explained on Section 4.5 are available since Qt 4.6 (released 1st November 2009 [10]).

For more detailed and up-to-date information of the features, the reader is referred to Qt Online Reference Documentation [12].

4.3.1 Signals and slots

Signals and slots is Qt's way to handle object communication. It is widely deployed and one of the first things encountered when going through the Qt tutorials. In short, a signal is emitted when an event occurs (for example, slider has been moved) and the slots react (adjust volume) on these events. Defining which slot reacts to which signal is done by connecting the signal to the slot. Thus connecting the move signal of the slider to the adjust volume slot creates a slider, which controls the audio volume. Signal can contain additional information, which is passed to slot as a member function parameter(s). Signal can also be connected to multiple slots or to another signal.

Signals are about ten times slower than traditionally used callbacks. Therefore the mechanism should not be used in some performance-critical places, such as to emit every single byte read from a disk. When used wisely, however, the performance difference to callbacks tends to be insignificant in real applications while giving the advantage of loose relations between objects. The emitter of a signal does not have

¹KDE e.V. is a non-profit organization representing KDE in legal and financial matters. *e.V.* stands for *eingetragener Verein*, which means registered association.

to know (or care) about the slots connected to the signal. The slot does not have to know if it is connected, either.

4.3.2 Meta-Object Compiler, `moc`

While meta-object compiler is not an actual feature, being familiar with it helps understanding other features.

In brief, `moc` is a program used to handle Qt's C++ extensions, such as signals and slots. It reads C++ files and produces a meta-object code of the classes having `Q_OBJECT` macro. The meta-object code is vital for signals and slots mechanism, run-time type information and dynamic property system. Moreover, `moc` does some additional code diagnostics like checks for illegal constructs. Unfortunately it places some limitations for the header files (`.h`), see [12].

Qt's build facility, `qmake`, runs `moc` and links `moc`-created code automatically. Therefore the developer rarely needs to call `moc` directly.

4.3.3 `QObject` trees

`QObject` trees are a way to organize `QObject`s. When organized as trees, the child objects are deleted when their parent is, thus easing the resource handling. Manual deletion with `delete` still is possible, because childs remove themselves from their parent when deleted. For example, windows are constructed of smaller areas (panels, bars, buttons, etc), and by "treeing" these objects the developer only has to worry about "important" objects. Following the example, when the window is closed (deleted), it automatically deletes all its children, effectively releasing all resources reserved in the tree for the window.

Object trees can also be walked up (`parent()`) and down (`children()`), and debugged (`dumpObject[Tree|Info]()`).

For `QObject`s created in stack (*i.e.* not allocated with `new`), care must be taken to construct the parent before its children. This way, the children are destructed (and removed from the parent) before the parent is. Doing the opposite leads to double destruction – that is, to problems.

4.3.4 `QObject` properties

Qt's property system is a compiler- and platform-independent library, based on Meta-Object System (MOS). The property system can be used to loosen the relations between `QObject`s the same way as with the signals and slots feature. Properties for `QObject`s are declared with `Q_PROPERTY()` macro, and collected with `moc`. Declared property behaves like a class data member, but can additionally be accessed through

MOS. The property can have different accessors, which are bound to the member functions:

- `READ` for reading the value.
- `WRITE` for setting the value.
- `RESET` for resetting to the default value.
- `DESIGNABLE` for property editor visibility (default `true`, visible).
- `SCRIPTABLE` for scripting access (default `true`, accessible).
- `STORED` to hint whether value is stored (`true`) or calculatable (`false`).
- `USER` to hint that (end-)user can edit this value (default `false`, deny).

After declaring a property, it can be read and set (when `WRITE` function is defined) with `QObject`'s `property(char* name)` and `setProperty(char* name, QVariant& value)` -functions, or straight with the bound member functions when the class is known at compile time.

New properties can be dynamically added at the runtime just by calling `setProperty` function with a new property name. If the given property is already defined, `setProperty` returns `true` or `false` whether the value can be set to the property type.

In addition to the properties, `QObject`s can also define invocable member functions with `Q_INVOKABLE` macro. When the definition of member function is prepended with the macro, the function can be invoked through `QObject::invokeMethod()`.

`QObject` properties can be used with Qt's ECMAScript interpreter, called Qt Script, which allows `QObject`s to be scripted with JavaScript. Qt makes `QObject`'s slots automatically available for use in the script (slots are publicly available for any class), and other variables and functions can be exported through the `QObject`'s properties.

4.3.5 QtNetwork module

Qt's networking facilities are provided by the `QtNetwork` module. This section provides an overview to the module, and shortly presents the capabilities of the "standard" Qt networking.

Qt has no POSIX-like API, but it wraps the socket functionality inside `QTcpSocket`, `QTcpServer`, and `QUdpSocket` classes. Of these, the `QTcpServer` presents the server functionality with TCP (for UDP, servers are implicit), and the rest

provide the transmission facilities. Both IPv4 and IPv6 are supported. SSL/TLS encrypted TCP sockets are available through `QSslSocket` class, with server functionality also available.

Besides the basic functionality, HTTP and FTP client functionality is supported through the class `QNetworkAccessManager`. Legacy classes, `QHttp` and `QFtp`, are also included, but `QNetworkAccessManager` is recommended over both for a simpler and more powerful API.

`QtNetwork` incorporates proxy functionality, which supports all previously mentioned classes. The functionality is offered with `QNetworkProxy` class. Proxy settings are defined to the class' instance, which is then set as the application-wide proxy with `QNetworkProxy::setApplicationProxy()`.

4.4 Mobility API

Mobile devices, like phones, often share some functionality. Most (if not all) phones on the market have a phonebook, where the user can store important contacts. To name another common function, phones also do networking over many bearers – most high-end phones today have WLAN, 3G and Bluetooth to choose from. These and many more functionalities are usually offered through custom platform-specific APIs, making cross-platform development a difficult and tedious job. This is where the Mobility API steps in.

Qt Mobility API is a suite of APIs targeted for mobile device functionality. It is currently announced for Symbian, Windows CE, and Maemo platforms, and provides an uniform, cross-platform API for functionality such as contact and bearer management. The Mobility API eases the development targeting mobile platforms, allowing developers to create one solution that can be deployed across multiple platforms. This is a significant advantage for mobile phone manufacturers, operators, and third party developers, who now can support multiple platforms with a single codebase.

The Mobility API already has some frameworks and APIs announced, which we will go through shortly to review the current possibilities for cross-platform mobile development.

- **Service framework** allows clients to discover and use services offered by other Qt applications. The discovery can be done based on service name and version, or its interface definition (where Qt's meta-object system assists). After the discovery, the framework starts the service and gives a pointer to it to the client. As in the most client-service schemes, a service can also use other services, thus playing a client and server role at the same time.
- **Contacts** provides a set of APIs for requesting contact data from local and/or

remote backends. The contact data is stored in a platform-specific manner, but the intention is to keep this hidden from Qt perspective and provide details in a platform and datastore independent manner.

- **Context framework** provides access to Qt value spaces. A value space is a single, consistent and tree-like datamodel. Applications can read the values, navigate the hierarchy and subscribe for value change notifications. For example, information about charge level and state of the device's battery can be provided to `/device/battery/charge` and `/device/battery/state`. Clients can then read the values and be signaled when they change.
- **Location** enables applications to be position-aware by providing location data. It incorporates different technologies such as GPS and cell ID positioning, and provides a latitude-longitude coordinate with a date and time.
- **Multimedia** provides a set of APIs to play and record media, and to manage a collection of media content.
- **Messaging** enables messaging services such as SMS, MMS, email and XMPP. Searching and sorting, sending and receiving, and composing of messages are supported.
- **Bearer management** controls the bearers used for networking. It is discussed in detail at the next section.

The Mobility API evolves with the (mobile) world and devices, and a more comprehensive list with newest APIs can be found from Qt Online Reference Documentation [12].

4.5 Bearer Management Library²

Bearer management in Qt is focused on providing applications an access to the desired (service) network. The bearer management library represents network connectivity of the underlying system to Qt applications, allowing them to control system's connectivity state. This state is used when application accesses the network (by creating a socket, for example). The library can be used for many purposes, including:

- Select the access point used by sockets.
- Get indications of roaming possibilities, and roam (or deny it).

²Bearer management library is a part of Qt Mobility API (discussed in Section 4.4), but has its own section due being central to the subject of this thesis.

- Find out system's connectivity capabilities and its connectivity configuration, and monitor it.

It is important to note that the bearer management library does not support configuration of networking settings. For example, it is not possible to change interface's IP or to add new WLAN access point configuration to system. Neither it handles network connections (the actual socket sessions) made by the application.

Bearer management library is also used by Qt's `QNetworkAccessManager` class.

To use this library, it is essential to understand two of its core concepts: network configuration and network session.

4.5.1 Network configuration

A network configuration is an abstraction of one or multiple access points. It represents a single access point or service network. However, it does not contain connection settings (like IP address, WLAN passwords or dialling number) and delegates actual connection creation, managing and closing to the access point it represents.

For example, a configuration named "University" (the naming is not defined by Qt) can be a WLAN access point. In this case, the system simply maps this configuration to the WLAN access point. The same configuration can also be a group of access points, containing WLANs, VPN configured over 3G and a modem connection. This is a "service network" in Qt terminology. When a network session using service network configuration is opened, the system selects an access point from the group by its own preferences.

The service network configurations are beneficial when a connection to a particular destination network is needed. Such network may not be available through every access point, for example the sample home network discussed earlier is not accessible through 3G –which is through the Internet– due to the router's firewall configuration. Special settings and procedures may also be needed when connecting to the network, which is the case with VPN. Such access point can be defined as a new access point to the service network.

Access point grouping is also a necessity for roaming in the service network, as other access points from the group can be used if some become unreachable or preferable over current access point. A common example of this is mobile browser changing bearer from WLAN to 3G, because the user walked out of effective WLAN area.

`QNetworkConfiguration` is Qt's class to contain the network configuration. It provides information about configurations' purpose (`Purpose`), state (`StateFlags`), and type (`Type`).

- `Purpose` specifies the purpose the configuration is for. For example, a `Private`

configuration is suitable for accessing a private network and a `Public` for the Internet.

- `StateFlags` specifies the state of the configuration. For example, a `Defined` configuration is properly set up but not currently usable (such as unplugged ethernet), and `Active` is already in use (maybe by some other application). This also applies to service networks, where a `Discovered` one means that one of the access points defined in service network is `Discovered` – and so on. The flags `Active`, `Discovered` and `Defined` are defined so that the former implies the latter.
- `Type` identifies `ServiceNetworks` and `InternetAccessPoints`. For the former, a list of access points is available through the method `children()`. There is also a special placeholder, `UserChoice`, to indicate that the *system* resolves the actual configuration. The system may consult the user, as the name implies.

`QNetworkConfiguration` has also a name, which can be shown to the user, and an identifier, from which the same configuration can be later constructed using `QNetworkConfigurationManager`.

4.5.2 Network session

Network session is a bearer management concept for selected network configuration. In the simplest case, the session is opened to a configuration having only one access point, and opening and closing the session starts and stops the access point together with the interface (bearer) it uses.

Network session may be used for roaming control if the configuration is a service network. For forced roaming, the platform informs the session when the roaming has happened. For application level roaming, the platform consults the session whether it wants to roam to a new configuration, or to continue using the current one. The support for roaming flavour is dependent of the platform and can be checked from system capabilities at run-time.

It is possible to have multiple sessions to different networks. Besides `QUdpSocket`, there is no way to bind a socket to the certain configuration or interface, which limits the use of this feature with Qt 4.5. However, when the bearer management API is migrated to Qt in the future, broader functionality will be supported.

`QNetworkSession` is bound to one `QNetworkConfiguration` and allows the application to use the configuration through `open` and `close` slots. If the platform has a `DataStatistics` capability, the application can also collect usage statistics of the configuration. With a signal `stateChanged()`, it is possible for the application to react on session's connectivity state changes, for example handle UI while roaming.

`QNetworkSession` is also an enabler for roaming. Every time a session roams to a new configuration, it emits the `newConfigurationActivated()` signal, allowing the session to either `accept()` or `reject()` the new configuration. Suitability of the new configuration can be examined at this point. By rejecting the new configuration, the old one stays in use. For more sophisticated roaming (ALR), application can connect to `preferredConfigurationChanged()` signal to consult the system about application's preferred configurations. The signal is automatically emitted whenever adequate, so no manual actions (like scanning) are needed to receive it. It should be noted that the system decides whether the newly found configuration is preferred before emitting the signal. This way, the system (or the user) can set the general priority of configurations.

Details regarding roaming can be queried through `QNetworkSession`'s properties. As the `configuration()` always returns the base configuration on which the session was constructed, `"ActiveConfiguration"` identifies the actual configuration used while roaming. `"UserChoiceConfiguration"` returns the configuration solved during opening, if the session was of type `UserChoice`. `"ConnectInBackground"` is a writable property, intended to be used by background applications. When set to `true` and a user consultation is needed, the connection fails instead, and the user is never consulted.

4.5.3 System capabilities

Different platforms have different capabilities for the bearer management and roaming. On some platforms, application may not allowed to start or stop interfaces, and on other, only partial roaming capabilities may be supported.

`QNetworkConfigurationManager` is Qt's class for querying system capabilities and network configurations. `QNetworkConfigurationManager` also signals the changes to the system's network configuration and connectivity state.

There are 6 different capabilities available through `capabilities()`. They are listed in the following.

- **BearerManagement** states that bearer control (start and stop of interfaces) is possible through `QNetworkSession`. If it is not possible, bearers may still be monitored and sessions can be opened, but the overall system's connectivity cannot be affected.
- **DirectConnectionRouting** is set when the IP level routing is overridden by interface selection. This means that when trying to reach the destination address, the packet is sent to the interface pointed by the session, rather than to the one pointed by the system's routing table. Direct routing generally

guarantees that the desired interface is used, and solves the case when two (or more) interfaces have conflicting IP ranges.

- `SystemSessionSupport` is `true` when the underlying platform guarantees that a interface is not closed until there is no session using it (by any process).
- `ApplicationLevelRoaming` indicates that the system consults applications in roaming. Additionally, applications need to connect to signal `preferredConfigurationChanged()` to be consulted.
- `ForcedRoaming` implies the support for automatic roaming, where the system decides which access point to use. Applications do not have any control over this process unless the `ApplicationLevelRoaming` flag is set. Roaming is signaled with `newConfigurationActivated()`. The new access point can be taken into use with `accept()`, in which case the sockets to the old access point may become unusable. Method `reject()` continues to use the current access point.
- `DataStatistics` means that data usage statistics are collected.

All and the default network configuration can be queried through `allConfigurations()` and `defaultConfiguration()`, respectively. Configuration update can be requested with `updateConfigurations()`.

`QNetworkConfigurationManager` also provides a way to track system's overall connectivity state through `onlineStateChanged()` signal.

4.6 HTTP level roaming

Qt provides a way to access the Internet and its services through `QNetworkAccessManager` class. The class works with `QNetworkRequests` and `QNetworkReplies`. Given the fact that HTTP is a stateless protocol, `QNetworkAccessManager` has the possibility to choose the best bearer to fulfill application's requests. For example, when a WLAN access point becomes available when using a 3G connection, `QNetworkAccessManager` can switch to it in the middle of using the connection and provide seamless roaming for the application without the application ever noticing anything. Thus, `QNetworkAccessManager` is an easy way for the applications to implement and benefit of HTTP optimized roaming. `QNetworkAccessManager` is included to `QtNetwork` module so no knowledge of mobility API is needed from the developer.

All the ease and simplicity does not come without a cost. Currently there is some shortcomings when using `QNetworkAccessManager`, most in the area of more serious bearer management:

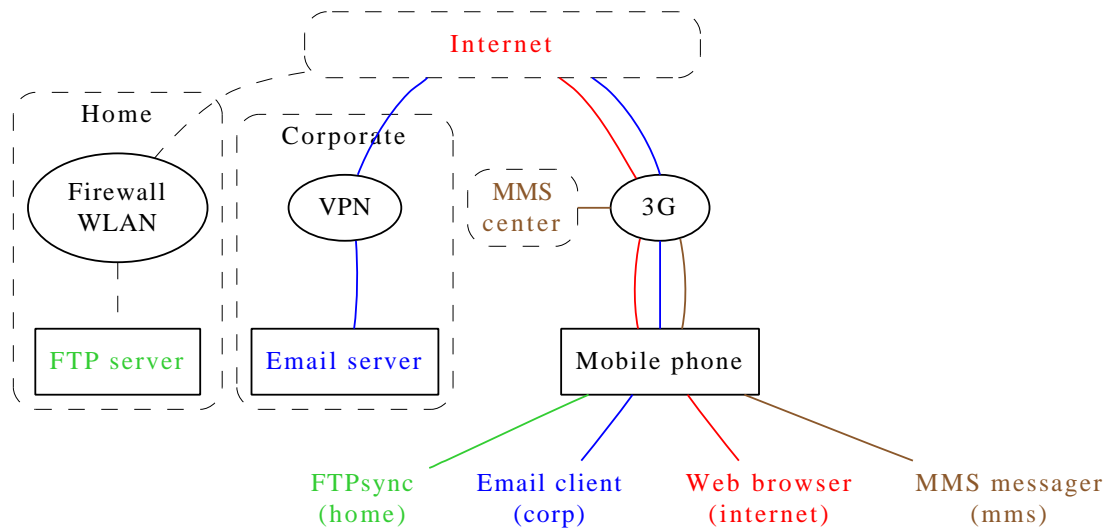


Figure 4.1: Sample mobile network with applications and service networks.

Table 4.3: Mobile networking example configuration

| Application | Service network | Access points |
|---------------|-----------------|--------------------------------------|
| FTPSync | Home | WLAN |
| Email client | Corporate | VPN (using Internet service network) |
| Web browser | Internet | WLAN, 3G |
| MMS messenger | MMS center | 3G |

- No support for automatic roaming in a service network *other than the Internet*. If the application wishes `QNetworkAccessManager` to roam on a service network, it needs to create and manage a `QNetworkSession` to it. When such session is opened, `QNetworkAccessManager` automatically uses the network made available by the session.
- No indication (signals) of roaming. If the application needs to know which access point is currently being used by `QNetworkAccessManager`, there is no way to find out. Without a session, it is also impossible to stop the access point (therefore saving power and costs) when the application knows it does not need the network connectivity for a while.

It should be noted that none of these shortcomings are fundamental and may be addressed in the future Qt versions.

4.7 An example of system mobility

To put together what we have discussed about Qt in this chapter, and of mobile networking in Chapter 3, we examine the behaviour of a mobile system with a two-phase scenario.

An example of mobile networking environment and configuration is presented in Figure 4.1 and Table 4.3. The home and corporate networks presented in the figure resemble the ones introduced in subsections 2.3.1 and 2.3.2, respectively. There are four applications, each having a different service network and network availability. The platform is assumed to support `BearerManagement` and `DirectConnectionRouting` capabilities.

- FTPsync sees the network as unreachable, and waits for the `QNetworkSession` to be opened. Because the session is to the Home network, FTPsync can not access the Internet if it wished, either.
- Email client can check the mail from corporate server, because the VPN tunnel can be built over the Internet, which is available through 3G. If, for some reason, the email client tries to reach the Internet, the traffic is tunneled through the VPN and then connected from the corporate network.
- Web browser is browsing the Internet with `QNetworkAccessManager`. 3G is automatically chosen by `QNetworkAccessManager`.
- MMS messenger uses 3G, too, because MMS center is also available via it.

Compared to the conventional networking, the first phase already reveals one difference. Because FTPsync has selected the currently unavailable service network, it cannot perform any networking (as defined by `DirectConnectionRouting` capability).

Continuing to the mobility phase, we assume that the user returns to home and the home WLAN becomes available. This gives different implications to the applications. The platform is assumed to be ALR capable.

- For the FTPsync, this means that the Home service network is finally reachable. FTPsync receives `sessionOpened()` and `stateChanged()` signals and may start transfers. The Internet is implicitly available through the home WLAN, so FTPsync can connect it, too.
- Email client does not know that the WLAN became available, because it is not listed on the Corporate's access points. The change is signaled to the VPN because it uses the Internet service network, where the WLAN is listed. VPN tests the WLAN and finds the needed ports unfiltered (not firewalled by the home router). Thus it accepts the preferred configuration (the WLAN access point). Depending of the implementation, connections through the VPN may suffer only short interruption or break totally, which is the only effect email client notices.

- Web browser does not know that the `QNetworkAccessManager` seamlessly starts to use WLAN, because it is being preferred over 3G. `QNetworkAccessManager` may even finish its pending transactions before roaming, losing no HTTP requests or replies.
- The change does not affect to MMS messenger, because the WLAN is not listed on its access points. It continues to use the 3G.

If automatic (forced) roaming is not utilized, the mobile phone is now in multi-homing state – it is connected to both WLAN and 3G networks. Of course, the MMS messenger can close its sessions when there are no messages to send or receive, thus disconnecting the 3G and making the system "single-homing" again.

If the automatic roaming is utilized, all applications would start using the preferred WLAN access point.

5. SAMPLE APPLICATION: FTPSYNC

To test and evaluate the bearer management API in practice, a basic networking application was implemented. Named as FTPsync, the application synchronizes a local directory with a remote one when a certain network is available. The user can define the target network (configuration in Qt terms), the FTP details (server, username, ...), the local and remote paths and the synchronization direction (from or to remote). When the network becomes available, FTPsync logs in to the FTP server and transfers nonexistent files to the target.

A practical operating environment for the application is the home network presented in Section 2.3.1. The user configures the target network to be the WLAN offered by the router, and sets the FTP details match the server. When the application is configured and started, and the user walks home, files are automatically synchronized through the WLAN.

Figure 5.1 shows the application running on Linux. The user is currently selecting the configuration to be `eth0`. The string `"5440722"` is unique identifier of the `eth0` configuration.

5.1 Architecture

Overall software architecture of FTPsync is presented in Figure 5.2. Functions, signals and slots central to the bearer management are also shown.

The application is basically divided in two parts, which are controlled by FTPsync main class. The other part manages the network and bearers, and the other the synchronization. The bearer management is visited in detail later.

Synchronization is largely based on `FileInfo` interface, which is a loose abstraction of filesystem and synchronization. The local filesystem is managed with classes derived from `QDir` and `QFileInfo`.

For FTP, Qt offers choices. `QFtp` and `QNetworkAccessManager` classes both provide FTP functionality. The latter was preferred by Qt documentation [12], but the former was chosen for two reasons. First, `QFtp` resembles sockets more closely as it represents a single FTP connection, where the `QNetworkAccessManager` is based on independent request-reply pairs. The connection-based approach gives greater visibility to the challenges of bearer management. Second, the documentation of `QNetworkAccessManager` mentioned that it can be used for FTP, but had no actual

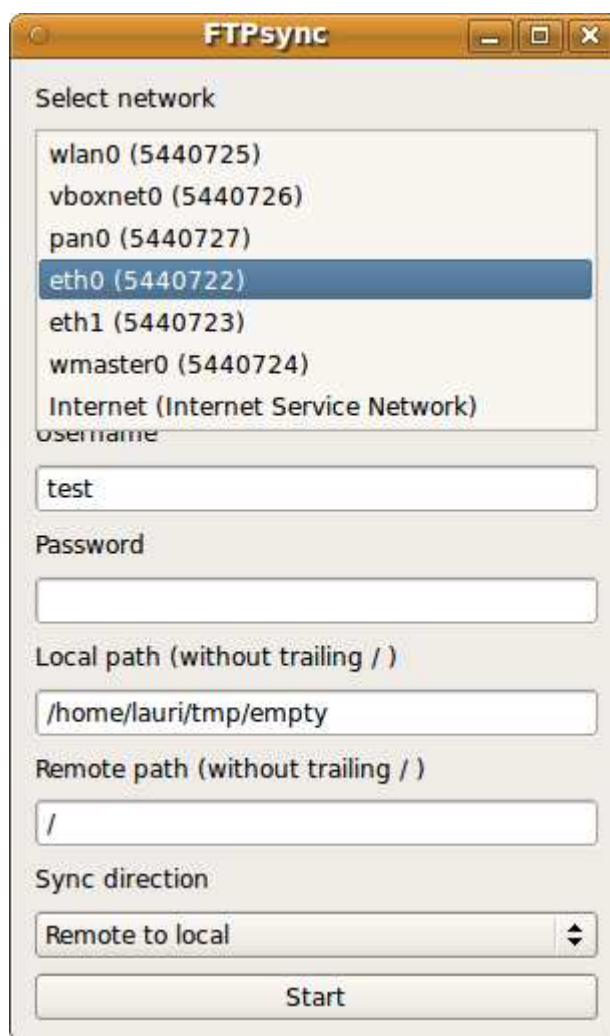


Figure 5.1: Configuration selection with FTPsync

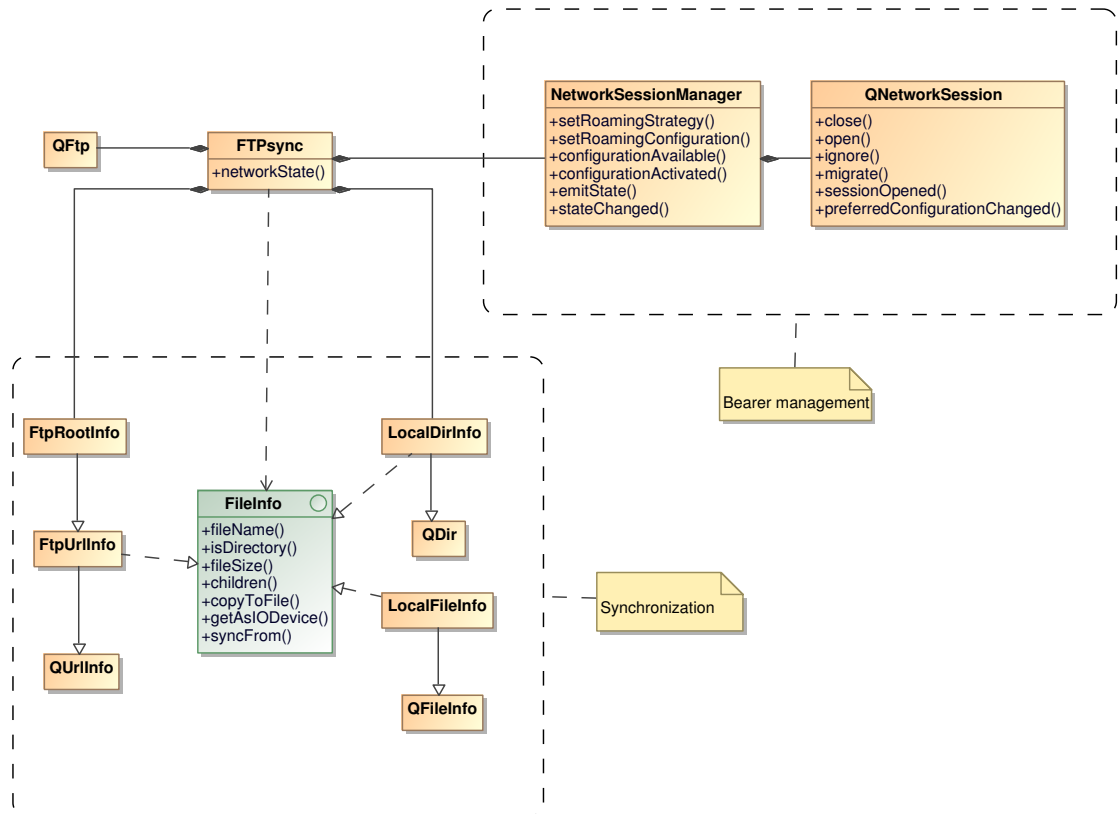


Figure 5.2: Architecture of FTPsync

documentation regarding how it is used with FTP.

5.2 Bearer management

Bearer management of the application is done by the `NetworkSessionManager` class, which handles one `QNetworkSession`. The class provides simplified view of the network state without application domain (FTP in this case) specific functionality. This approach was selected to keep the functionality of the sample code as simple and widely usable as possible.

For implementation details, the source code for `NetworkSessionManager` class (files `NetworkSessionManager.h` and `NetworkSessionManager.cpp`) is listed on Appendix A.

5.2.1 Behavior

The application can set the desired roaming strategy reflecting wanted network behaviour. There are three roaming strategies (`NetworkSessionManager::RoamingStrategy`).

- **Offline**, which means that the application does not need the network connectivity at the moment. This is the default state, and other states implicitly indicate that the application wants to use the network.
- **BestAvailable** indicates that the application wants the best available network connectivity. The best in this case is the most preferred by system's or user's preferences, as the functionality is not specific to any application-domain. There may be open connections, but it does not matter if they break.
- **MaintainConnections** is used, when the current connections should be maintained as long as possible. This is done by denying roaming in cases it would not be seamless. Of course, it is impossible to completely guarantee that the established network connections do not break.

The `NetworkSessionManager` signals when the network connectivity state changes through the signal `stateChanged(NetworkSessionManager::NetworkState)`. By connecting to the signal, the application can trigger actions related to the state, such as reconnects.

- **NetworkOffline**, which indicates that the network can not be connected.
- **NetworkOnline**, which means that the network can be connected.
- **NetworkError**, which means that a network error has happened. There is no way to retrieve the error.

Once the synchronization is activated (user presses **Start** button), `FTPSync` builds the `NetworkSessionManager` with the selected configuration, connects to the `stateChanged` signal and sets the kind of connectivity behaviour is needed. The application then acts according to the connectivity state signaled by the manager. When `NetworkSessionManager` emits **NetworkOnline**, an ftp connection is established (implicitly using the configuration managed by the session in the `NetworkSessionManager`) and then used for synchronization. Additionally, the manager is set to **MaintainConnections** in order to avoid roaming during synchronization.

5.2.2 Interaction with bearer management library

Sequence diagrams of some central actions are presented here to get a grasp on how the `FTPSync`'s bearer management works.

Opening a network session

Figure 5.3 represents the opening of a network session. `FTPSync` sets the configuration to `NetworkSessionManager`, which creates a corresponding session. Signals

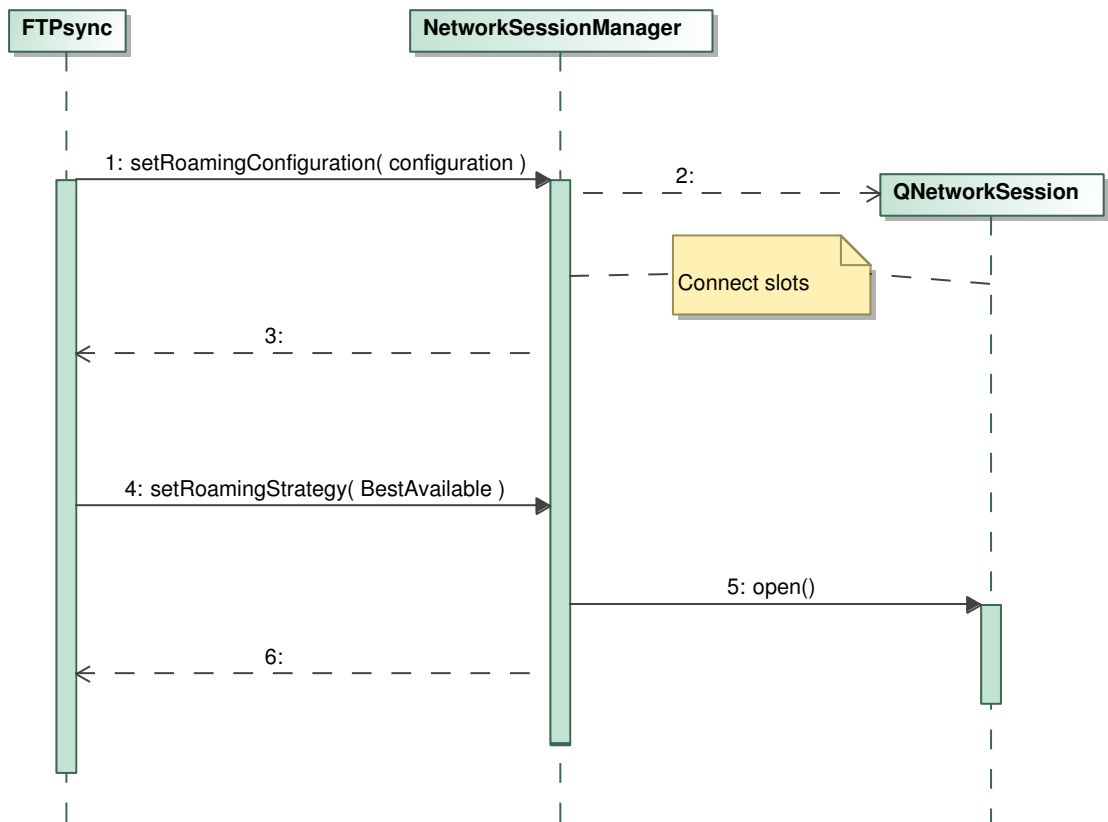


Figure 5.3: Opening a network session

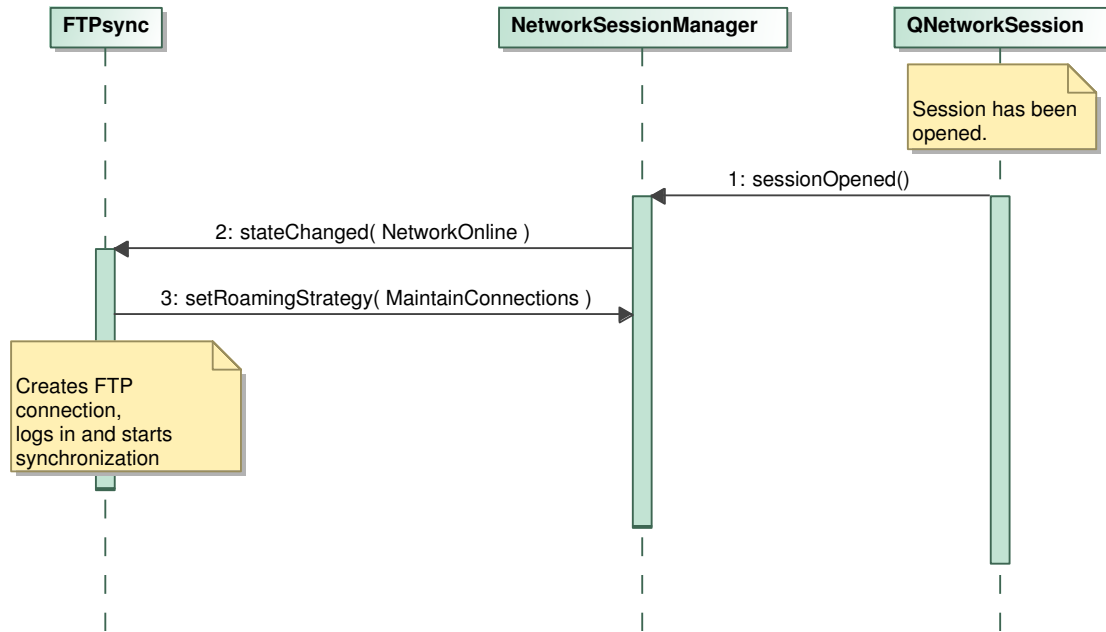


Figure 5.4: Starting a transfer

`sessionOpened`, `sessionClosed`, `error` and `preferredConfigurationChanged` from the created session are connected to the manager. After the configuration is set, **FTPsync** sets the roaming strategy to `BestAvailable`, which leads the managed session to be opened. If preferred configurations become available, they are taken into use. This way the FTP connection has the most preferred configuration in use once started.

Starting a transfer

Figure 5.4 shows the events leading to the synchronization. The network session has been requested to be opened. When the opening has finished, the session emits `sessionOpened`, which the manager translates to `NetworkOnline` state. The state is emitted to **FTPsync**. Before initiating FTP connection, **FTPsync** sets the roaming strategy to `MaintainConnections` in order to prevent connection breaks.

Roaming

Figure 5.5 goes through the choices made when a preferred configuration becomes available, which is the case of application level roaming. For `BestAvailable` roaming strategy, `migrate` is called always.

When the roaming strategy is set to `MaintainConnections`, the decision is based on the current session state and the "seamlessness" of the roaming. When seamless roaming is possible, the session can `migrate` because the connections should not break. Also, if the session is not currently active, there should not be connections

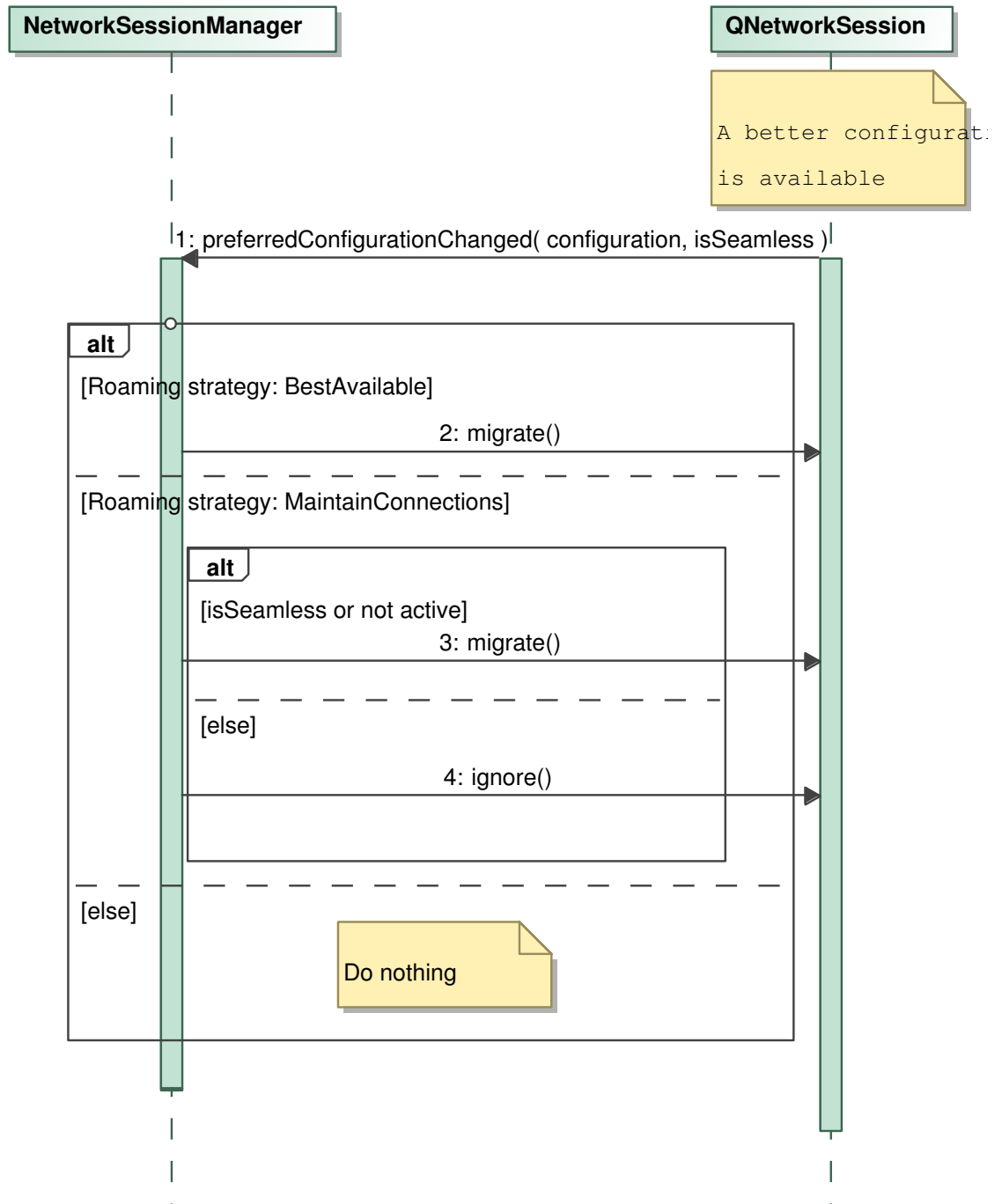


Figure 5.5: Application level roaming

that break, so the session can migrate. Otherwise the only option is to ignore the configuration.

For automatic (forced) roaming and `newConfigurationActivated` signal, the actions are similar but `accept` and `reject` instead of `migrate` and `reject`, respectively. The activated configuration is assumed seamless and is not tested for connectivity.

The roaming process is invisible to `FTPSync`, as it does not change the state of the network connectivity.

5.2.3 Application specific management

As the examples in the previous section discussed generic bearer management, this section takes a look on how to tailor the bearer management to the needs of certain application (and protocol). The `NetworkSessionManager` implementation and the architecture of the `FTPSync` application are used as the starting point for improvements.

`NetworkSessionManager` does not test the suitability of a new configuration when it gets activated. For a general case, the destination host could be ping'd¹ to evaluate network connectivity. If the destination address and port is known, a TCP connection may be established for this purpose.

For protocols transmitting sensitive information unencrypted and without mutual endpoint authentication, it may be justified to deny the networking when `DirectConnectionRouting` is not supported by the platform. When the feature is not supported, basically any available bearer may be used to reach the target server and with an unexpected routing choice², the sensitive information may get revealed. The FTP protocol is one such example, and a fake FTP server at the destination IP on the "wrong" bearer can extract the plaintext username and password information the client sends when logging in. Supported `DirectConnectionRouting` eliminates the ambiguous packet routing, as the packets are always sent through the intended configuration (access point).

An architectural improvement could be to change the ownership of the connection. In `FTPSync` (Figure 5.2), the `ftpsync` main class owns the connection and gives it to the FTP classes. Because `NetworkSessionManager` provides generic network management and the application is such a simple one, this is not a big issue. However, if the connection was owned by `NetworkSessionManager` (acting as a connection factory or FTP command dispatcher), it would allow more fine-grained

¹`ping` command tests if a host is reachable across the network with an ICMP message. However it does not guarantee that certain protocol or port is allowed in the network. Ping can not be done solely within Qt as ICMP is not supported.

²Traffic is not routed through the access point expected by the application.

control over the network usage. For example, when a new preferred configuration comes available, the manager can create a new FTP connection through the configuration and start using that, eventually leading to FTP "level" roaming. There is also FTP command `PORT`³ to be (mis)used for an ultimate FTP multi-homing experience.

There may be application specific restrictions too. Particularly with FTP, the servers often limit the number of concurrent connections on server or user basis. After aggressive roaming, there may be enough dangling connections to hit the limit, preventing new logins until the "danglers" time out.

5.3 Conclusions from FTPsync

First of all, the bearer management API was straightforward and effective to use. It took a class and handful of functions (2 member functions, 3 slots, and 1 signal) to implement a basic network manager, featuring support for automatic and application level roaming. The API was also further improved as a result of discussions between the author and the manager of the API, making the use of `QNetworkSession` class more straightforward, and clarifying the aspects of bearer monitoring and control in documentation [12].

The piece missing from the bearer management puzzle is proper support for multiple `QNetworkSessions`. While the sessions can be created and opened without limitations, there is no way to define which one (of the active ones) will be used for a socket. This missing piece will be supported once the bearer management API migrates into Qt.

Ideally every platform would have an adequate support for bearer management. In practice, this support varies considerably between platforms. Currently the Symbian platform enjoys the most comprehensible feature set, while others provide some, but not all of the features. The most common shortcoming is the support for roaming. The differing support poses challenges for the development. In order to develop support for advanced functionality such as roaming, the development must be done on a platform supporting it. This may add development costs. Additionally, the functionality of the code on platforms with reduced feature support should be tested.

From programming perspective, the following observations were made of the Qt framework. The signals and slots feature was nice, and really decoupled things well. Also, the `QObject` tree feature was excellent, and it did not took a long to notice that parents were set for every object created. With a bit of practice, the created tree started to look like a proper one – every object is in the right place, under the right parent. Generally this create-place-forget-pattern eased and loosened the

³`PORT` specifies the host and port to which the server should connect for the next file transfer.

resource handling.

There also was some oddities. For example, `QComboBox` allowed to add (`QString& text`, `QVariant& userdata`) pairs into it, but for some reason, it had no `activated(QVariant&)` signal. Instead, one had to connect to `activated(int index)` and manually resolve the `userdata` (to be used as a configuration identifier string) from the box. Also, a proper documentation about usage of `QNetworkAccessManager` with FTP was missing. One can guess that files might be retrieved by making a request to an FTP url (`ftp://user:password@host/file`), but for file listings, the guess is not so easy.

6. CONCLUSION

We started this thesis with the art of protocols, services and networks and gave it a name of conventional networking. Basic building blocks – standards such as TCP/IP model, HTTP and POSIX API combined with the network components of varying scale – gave us a solid base for networking and information sharing.

Then we moved to the mobile networking, quickly noticing how similar, yet still different it is. The basic building blocks from conventional networking are mixed with new challenges and aspects: the usage of third-party networks affect to the cost structure, connectivity and security. Power consumption of the networking is an important factor, as the mobile devices are battery-powered. Bearer management and roaming decisions need to be made by the user, the system or applications – which gives the latter a chance to tailor the network behavior by its specific needs. And the new aspect of grouping access points to a service networks allows an access to the network on the basis of the services needed. We also discussed the question of how the basic blocks fit to the new challenges and aspects – the answer was moderately and insufficiently.

The search for better answer continued as we entered to the world of Qt. This featureful cross-platform C++ framework has Mobility API suite covering features often encountered in mobile platforms (calendar, messages, multimedia, location, and so on). The API also contains bearer management library suited for our needs. It represents networking as a session to (service) network, which can control the network configuration (bearer) and roam when a preferred one is found. The approach is easy to understand and powerful to use. There is also automatic HTTP level roaming built into the standard Qt since version 4.7.

The bearer management framework was also experimented with a sample application. A basic network manager class was implemented. The main functionality was expressed as sequence diagrams, too. The basic improvements to architecture and functionality of application-specific bearer manager was discussed. The usage of the bearer management API was simple and straightforward, and it required only a handful methods to create a basic, roaming session manager. The API was also improved based on the discussions during the experiment. As the platform capabilities vary, full support for some networking features may be available only on certain platforms.

REFERENCES

- [1] Blanchette, J., Summerfield M. 2006. C++ GUI programming with Qt4, First edition. Prentice Hall. ISBN 0-13-187249-4.
- [2] Comer, D. 2000. Internetworking with TCP/IP, 4th ed. Prentice Hall. ISBN 0-13-018380-6.
- [3] Fielding, Roy T., Taylor Richard N. 2002. Principled Design of the Modern Web Architecture. Referred 15.12.2009. Available at <http://www.ics.uci.edu/~taylor/documents/2002-REST-T0IT.pdf>
- [4] IEEE Standard 1003.1, 2004 Edition. Referred 15.12.2009. Available at <http://www.opengroup.org/onlinepubs/009695399/>
- [5] International Telecommunication Union. 1994. ITU-T Recommendation X.200: Open Systems Interconnection - Basic Reference Model: The basic model. Referred 29.12.2009. Available at <http://www.itu.int/rec/T-REC-X.200/recommendation.asp?lang=en&parent=T-REC-X.200-199407-I>
- [6] KDE Free Qt Foundation. Referred 15.12.2009. Available at <http://www.kde.org/whatiskde/kdefreeqtfoundation.php>
- [7] KDE - History: The Qt Issue. Referred 29.12.2009. Available at <http://www.kde.org/whatiskde/qt.php>
- [8] Nokia enriches application development with Qt for S60. Referred 15.12.2009. Available at <http://www.nokia.com/press/press-releases/showpressrelease?newsid=1261033>
- [9] Nokia to increase adoption of Qt with additional licensing option. Referred 15.12.2009. Available at <http://www.nokia.com/press/press-releases/showpressrelease?newsid=1281996>
- [10] Nokia Releases Qt 4.6. Referred 29.12.2009. Available at <http://qt.nokia.com/about/news/nokia-releases-qt-4.6>
- [11] Qt Licensing. Referred 15.12.2009. Available at <http://qt.nokia.com/products/licensing>
- [12] Qt Online Reference Documentation. Referred 15.12.2009. Available at <http://qt.nokia.com/doc/>
- [13] RFC 791. Internet Protocol. Referred 15.12.2009. Available at <http://www.ietf.org/rfc/rfc791.txt>

- [14] RFC 1122. Requirements for Internet Hosts – Communication Layers. Referred 15.12.2009. Available at <http://www.ietf.org/rfc/rfc1122.txt>
- [15] RFC 2460. Internet Protocol, version 6 (IPv6) Specification. Referred 15.12.2009. Available at <http://www.ietf.org/rfc/rfc2460.txt>
- [16] RFC 2616. Hypertext Transfer Protocol – HTTP/1.1. Referred 15.12.2009. Available at <http://www.ietf.org/rfc/rfc2616.txt>
- [17] RFC 3344. IP Mobility Support for IPv4. Referred 15.12.2009. Available at <http://www.ietf.org/rfc/rfc3344.txt>
- [18] RFC 3775. Mobility Support in IPv6. Referred 15.12.2009. Available at <http://www.ietf.org/rfc/rfc3775.txt>
- [19] Trolltech and KDE Free Qt Foundation Announce Amended and Restated Software License Agreement. Referred 15.12.2009. Available at http://www.kde.org/whatiskde/kdefreeqt_announcement_20040723.php
- [20] World Wide Web Consortium (W3C). 2007. SOAP Version 1.2 Part 1: Messaging Framework. 2nd edition. Referred 29.12.2009. Available at <http://www.w3.org/TR/soap12-part1/>
- [21] World Wide Web Consortium (W3C). Web Services Activity home page. Referred 29.12.2009. Available at <http://www.w3.org/2002/ws/>

A. NETWORKSESSIONMANAGER IMPLEMENTATION

Listing A.1: Simplified NetworkSessionManager.h

```

#include <QObject>
#include <QNetworkSession>

class NetworkSessionManager: public QObject {
    Q_OBJECT
public:
    enum RoamingStrategy {
        Offline , BestAvailable , MaintainConnections
    };
    enum NetworkState {
        NetworkOffline , NetworkOnline , NetworkError
    };
    NetworkSessionManager( QObject* parent );
    ~NetworkSessionManager();
    void setRoamingStrategy( RoamingStrategy strategy );
    void setRoamingConfiguration(
        QNetworkConfiguration &configuration );
public slots:
    void configurationAvailable(
        const QNetworkConfiguration& config ,
        bool isSeamless );
    void configurationActivated ();
    void emitState ();
signals:
    void stateChanged(
        NetworkSessionManager::NetworkState newState );
private:
    QNetworkSession* session_ ;
    RoamingStrategy strategy_ ;
    NetworkState state_ ;
};

```

Listing A.2: Simplified NetworkSessionManager.cpp

```

#include "NetworkSessionManager.h"

NetworkSessionManager::NetworkSessionManager(QObject* parent)
    : QObject( parent ), session_( NULL ),
      strategy_( Offline ), state_( NetworkOffline ) { }

NetworkSessionManager::~NetworkSessionManager() { }

void NetworkSessionManager::setRoamingStrategy(
    RoamingStrategy strategy ) {
    switch ( strategy ) {
    case Offline:
        session_ ->close ();
        break ;
    case BestAvailable:
    case MaintainConnections:
        session_ ->open ();
    }
    strategy_ = strategy;
}

void NetworkSessionManager::setRoamingConfiguration(
    QNetworkConfiguration &configuration ) {
    if ( session_ != NULL ) { session_ ->deleteLater (); }
    session_ = new QNetworkSession( configuration ,
        this );
    connect( session_ , SIGNAL( sessionClosed() ) ,
        this , SLOT( emitState() ) );
    connect( session_ , SIGNAL( sessionOpened() ) ,
        this , SLOT( emitState() ) );
    connect( session_ , SIGNAL( error() ) ,
        this , SLOT( emitState() ) );
    connect( session_ , SIGNAL( preferredConfigurationChanged(
        const QNetworkConfiguration&, bool ) ) ,
        this , SLOT( configurationAvailable(
            const QNetworkConfiguration&, bool ) ) );
    connect( session_ , SIGNAL( newConfigurationActivated() ) ,
        this , SLOT( configuratinActivated() ) );
}

```

```

}

void NetworkSessionManager::configurationAvailable(
    const QNetworkConfiguration& config,
    bool isSeamless ) {
    switch ( strategy_ ) {
    case BestAvailable:
        session_ ->migrate();
        break;
    case MaintainConnections:
        if ( isSeamless || !session_ ->isActive() )
            { session_ ->migrate(); }
        else { session_ ->ignore(); }
    }
}

void NetworkSessionManager::configurationActivated() {
    switch ( strategy_ ) {
    case BestAvailable:
        session_ ->accept();
        break;
    case MaintainConnections:
        if ( !session_ ->isActive() )
            { session_ ->accept(); }
        else { session_ ->reject(); }
    }
}

void NetworkSessionManager::emitState() {
    NetworkState state;
    if ( session_ ->error() )
        { state = NetworkError; }
    else if ( session_ ->isActive() )
        { state = NetworkOnline; }
    else { state = NetworkOffline; }
    if ( state != state_ ) {
        state_ = state;
        emit stateChanged( state ); }
}

```
