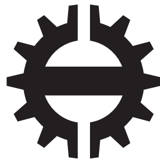

TAMPERE UNIVERSITY OF TECHNOLOGY

ANTTI LAINE

User Interface Prototypes for Social Ad Hoc Networking

Master of Science Thesis

Examiners: Adjunct prof. Marko Hännikäinen
Dr. Tech. Jukka Suhonen
Examiners and topic approved by the
Faculty Council of the Faculty of
Computing and Electrical Engineering
on 7.11.2012

PREFACE

It has been very interesting to be a part in studying and also actually creating a completely new social application and to be able to see it in use of hundreds of people.

I would like to thank everyone who have participated on the TWIN project and made it and this thesis possible: my coworkers at DCS and people at NRC Helsinki. I would also like to thank Marko Hännikäinen ja Jukka Suhonen for their valuable advice in writing this thesis.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
LAINE, ANTTI: User Interface Prototypes for Social Ad Hoc Networking
Master of Science Thesis, 61 pages
December 2012
Major: Programmable platforms and devices
Examiners: Adjunct prof. Marko Hännikäinen, Dr. Tech. Jukka Suhonen
Keywords: User interfaces, prototyping, social networks, ad hoc networks

WLAN technology available in everyday handheld devices conforming to the IEEE 802.11 standard includes ad hoc networking mode. The use of ad hoc networking enables devices to communicate directly with each other, without the need for external infrastructure or Internet connection. The technology has not gained popularity, as there has been a lack of a driving application.

This thesis presents user interfaces and use cases for TWIN, a social application using mobile ad hoc networking. TWIN has been developed at the Tampere University of Technology, Department of Computer Systems, in cooperation with Nokia Research Center. TWIN enables users to find other physically nearby users, form social groups, share media and send messages.

This thesis also presents methods used to accomplish fast prototyping and iterative development to be able to experiment with new features and ideas. Problems related to evaluation and testing of graphical user interfaces are also discussed.

The main results of the thesis are the design of use cases and user interfaces, and the implementation of those interfaces in TWIN. Secondary results presented here are the evaluation of the use cases and user interfaces, and the evaluation of methods used for fast prototyping and testing of the application.

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
LAINE, ANTTI: Sosiaalisen ad hoc -verkkosovelluksen käyttöliittymäprototyypit
Diplomityö, 61 sivua
Joulukuu 2012
Pääaine: Ohjelmoitavat alustat ja laitteet
Tarkastajat: Dosentti Marko Hännikäinen, TkT. Jukka Suhonen
Avainsanat: Käyttöliittymät, prototyypit, sosiaaliset verkot, ad hoc -verkot

Tavanomaisissa kämmenlaitteissa käytössä oleva WLAN-teknologia sisältää IEEE 802.11 -standardin mukaisen ad hoc -yhteystilan. Sen käyttö mahdollistaa laitteiden keskustelemisen välittömästi toistensa kanssa, ilman tarvetta ulkoiselle infrastruktuurille tai Internet-yhteydelle. Tekniikka ei kuitenkaan ole saavuttanut suosiota johtuen hyödyllisten sovellusten puutteesta.

Tämä diplomityö esittelee TWIN-sovelluksen käyttöliittymät ja käyttötapaukset. TWIN on Tampereen teknillisen yliopiston tietokonetekniikan laitoksella yhdessä Nokia Research Centerin kanssa toteutettu sosiaalinen sovellus, joka käyttää hyväkseen kommunikaatiota ad hoc -verkkojen välityksellä. TWIN mahdollistaa käyttäjien löytää muita välittömässä fyysisessä läheisyydessä olevia käyttäjiä, muodostaa sosiaalisia ryhmiä, jakaa mediaa ja lähettää viestejä toisilleen.

Tämä diplomityö esittelee myös nopeaan prototyypitykseen ja iterointiin käytettyjä menetelmiä, jotka mahdollistivat uusien toimintojen ja ideoiden nopean kokeilemisen. Lisäksi diplomityö käsittelee graafisen sovelluksen testaamiseen ja arviointiin liittyviä ongelmia.

Diplomityön pääasiallinen tulos on käyttötapausten ja käyttöliittymien suunnittelu ja toteuttaminen TWIN-sovelluksessa. Toissijaisia esiteltäviä tuloksia ovat käyttötapausten ja käyttöliittymien evaluointi, sekä nopeaan prototyypitykseen ja sovelluksen testaamiseen käytettyjen metodien evaluointi.

CONTENTS

1. Introduction . . . 1
2. Requirements and constraints for prototype application . . . 3
   2.1. Requirements for user interfaces . . . 3
      2.1.1. Ease of use . . . 3
      2.1.2. Scalability for large number of users . . . 4
      2.1.3. Ability to interact with other users . . . 4
      2.1.4. Ability to find other users . . . 4
      2.1.5. Ability to share information and media . . . 4
   2.2. Constraints of the platform . . . 4
      2.2.1. Maemo platform on Nokia N900 mobile computer . . . 5
      2.2.2. Ad hoc WLAN . . . 6
      2.2.3. Scalability constraints from the platform . . . 8
      2.2.4. Touch screen interface . . . 8
      2.2.5. User interface constraints . . . 8
   2.3. Related work . . . 9
3. Use cases for TWIN . . . 11
   3.1. View nearby user . . . 13
   3.2. Follow other users . . . 14
   3.3. Create communities . . . 15
   3.4. Join communities . . . 16
   3.5. Share personal information . . . 17
   3.6. Sharing media . . . 19
   3.7. Chat with other users . . . 20
   3.8. Post messages . . . 21
   3.9. Expressing mood . . . 22
4. Architecture of TWIN . . . 23
   4.1. Plugin interface . . . 23
   4.2. Core / UI separation . . . 23
   4.3. Architecture . . . 25
5. Implementation . . . 27
   5.1. Methods . . . 27
      5.1.1. Fast iterations . . . 28
      5.1.2. Immediate testing . . . 29
      5.1.3. Tagging commits . . . 29
   5.2. Tools . . . 31
      5.2.1. Git . . . 32
      5.2.2. Python . . . 32

5.2.3. GTK+ 34
5.2.4. Portability of Python and GTK+ 34
5.3. Plugins 35
5.3.1. register_plugin 35
5.3.2. get_plugin_by_type 35
5.3.3. ready 35
5.3.4. gui_init 35
5.3.5. cleanup 36
5.3.6. user_appears, user_disappears, user_changes 36
5.3.7. community_changes 36
5.3.8. Example plugin 36
6. User interface prototypes 38
6.1. Community view 38
6.2. Profile view 40
6.3. Radar view 41
6.4. Filesharing 43
6.5. Chat 45
6.6. Message board 47
6.7. Summary of the prototype interfaces 48
7. Prototype testing 49
7.1. Methods 49
7.1.1. PC version 49
7.1.2. Assertions 52
7.1.3. The Python Debugger 52
7.2. Problems 53
7.2.1. Interpreted and dynamic 53
7.2.2. User interface testing 54
8. Evaluation 55
8.1. Usability of user interfaces 55
8.1.1. Ideas for improvement 55
8.2. Methods 56
9. Conclusions 57

LIST OF FIGURES

2.1. Nokia N900 mobile computer. 5
2.2. Example of a TWIN network. 7
2.3. Maemo 5 home screen. 9

3.1. Use cases for TWIN. 12
3.2. Activity diagram for viewing users use case. 13
3.3. Activity diagram for following users use case. 14
3.4. Activity diagram for creating community use case. 15
3.5. Activity diagram for joining communities use case. 16
3.6. Activity diagram for sharing personal information use case. 18
3.7. Activity diagram for sharing media use case. 19
3.8. Activity diagram for chatting with other users use case. 20
3.9. Activity diagram for posting messages use case. 21
3.10. Activity diagram for expressing mood use case. 22

4.1. TWIN core uniform plugin interface. 24
4.2. Architecture design of TWIN as a stack diagram. 25

5.1. Development flow of TWIN. 28

6.1. Community view. 39
6.2. Inside twin community. 39
6.3. Personal profile. 40
6.4. Profile editor. 41
6.5. Radar view. 42
6.6. Filesharing adding files view. 43
6.7. Filesharing view. 44
6.8. Filesharing download sequence. 45
6.9. Chat. 46
6.10. Message board. 47

7.1. Timeline for evaluation. 49
7.2. TWIN running on a PC. 51
7.3. Python debugger. 53

LIST OF TABLES

2.1. Specifications of N900. 6
2.2. Summary of related works. 10

5.1. Tags for different commit types. 30
5.2. Custom tools used in development of TWIN. 31
5.3. Most used Python Standard Library modules in TWIN. 33
5.4. Summary of used 3rd party libraries. 34

6.1. Main user interfaces implemented for TWIN. 48

7.1. Differences of automated error checking. 54

8.1. Most interesting features of TWIN according to pilot participants. . . 56
8.2. Ideas for new features and use cases gathered from the pilot. 56

LIST OF PROGRAMMES

5.1. Simplest possible plugin. 37
5.2. Simplified example of plugin that vibrates the device, when a friend joins the network or a while is received. 37
7.1. Example of detecting the run-time platform and creating a different button widget depending of the result. 50

TERMS AND ABBREVIATIONS

Ad hoc Literally *for this (purpose)*, in WLAN a technology for enabling peer to peer communication without infrastructure

AST Abstract Syntax Tree, tree presentation of source code

Bencode A lightweight encoding for serialization of basic Python data structures

Bluetooth Short range wireless communication technology

C An imperative programming language

C++ An object oriented programming language, originated from C

CPU Central Processing Unit

DHT Distributed Hash Table, a service lookup technique using hash tables, that are distributed between multiple peers

Duck typing Style of dynamic typing, where objects are used according to their properties instead of their types

EDGE Enhanced Data rates for GSM Evolution, an extension for GPRS

GTK+ GIMP ToolKit, a cross-platform UI toolkit

GPRS General Packet Radio Service, a second generation packet oriented mobile data service for cellular networks

Hop Term used to describe passing traffic between two directly connected peers in a multi-hop network

Hop count Number of hops when routing traffic in a multi-hop network

HSPA High Speed Packet Access, a 3.5th generation mobile data service protocol

IPC Inter-process communication, set of techniques for different processes to communicate with each other

Interpreter A program which directly executes source code, as opposed to compiling it to machine readable instructions

Linux An open-source operating system kernel

LOC Lines of code

Maemo A Linux-based operating system for handheld computer

MeeGo An operating system for handheld computers, based on Maemo

Model–view–controller Programming pattern for separating core and user interface functionality

Multi-hop routing protocol A protocol that routes traffic over multiple peers

Observer pattern See publish–subscribe

pdb Python Debugger

PPI Pixels Per Inch, the number of pixels on an 1 inches wide and 1 pixels high line

Publish–subscribe Programming pattern for asynchronously forwarding data to receivers

Python An interpreted dynamically typed programming language

Qt An application framework

QWERTY Keyboard layout used in most computers

Reachability Usually a possibility to establish a connection. Here defined as the possibility to establish a direct connection between two devices for the purpose of downloading media.

RPC Remote Procedure Call, a method for requesting actions from a remote process

Signals and slots Publish–subscribe implementation used in Qt

Singleton Object that has exactly one instance

TWIN Although written in all capitals, TWIN is not an abbreviation

UI User interface

UML Unified Modeling Language, a standard for visually expressing structure of software systems

WCMDA Wideband Code Division Multiple Access, a third generation packet interface for cellular networks

WLAN Wireless Local Area Network

1. INTRODUCTION

Wireless links between mobile computers usually require an Internet connection and a server. This kind of connectivity is dependent on the availability of needed infrastructure. Devices can also communicate directly in a peer-to-peer manner. One technology to accomplish this is the ad hoc mode of WLAN communication, defined in IEEE 802.11 [31] set of standards.

Although ad hoc communication has been available since the first version of the IEEE 802.11 standard, it has rarely been used because of the lack of a driving application. As there has not been much practical use for the technology, its popularity has remained low. [27]

As connections between devices using this technology are only possible with direct links, the devices active in the network must be at that moment in close physical proximity. Because the connection does not depend on external infrastructure, applications utilizing ad hoc technology can be used in any place imaginable.

In this thesis, social local ad hoc networking application is defined as a software offering social interaction between users, and benefiting from the use of ad hoc WLAN by using direct links between devices in the local geographical area.

This thesis presents the design of use cases and user interfaces developed for TWIN, a social local ad hoc network application. TWIN has been developed as a prototype for the Nokia Instant Community concept. This thesis is a part of the TWIN project at the Tampere University of Technology (TUT), Department of Computer Systems (DCS). TWIN project is a research project between TUT and Nokia Research Center, started in June, 2008.

TWIN was developed for the Nokia N900 mobile device using a Linux-based Maemo operating system. TWIN has then been published as open source with a permissible modified BSD license with the name Proximate [52].

A social application was selected to provide a new type of useful and interesting services which can be used to interact people. Social aspect relates naturally to the concept of locality and physical closeness of the users.

The scope of this thesis is to present the development of a social application from design of use cases and user interfaces to a usable prototype and to discuss and evaluate the methods used in the TWIN project to enable rapid prototyping without formal specifications.

Work of the author for this thesis consists of designing the user interfaces in cooperation with other developers in the project, formulating the use cases from these designs, and implementing them, partly continuing from the work of other developers. Plugin interface presented in the thesis is work of others, and is utilized by the interfaces. Core functionality, which is outside of the scope of this thesis, is also work of other developers.

The thesis is divided as follows. Requirements for the prototype application dictated by the target platform are discussed in Chapter 2. Use cases for TWIN are presented in Chapter 3. General architecture, as well as the plugin interface architecture are presented in Chapter 4. Methods and tools of implementation are presented in Chapter 5. User interface prototypes are presented in Chapter 6. Methods and problems for testing of a graphical application written in a dynamically typed interpreted language are discussed in Chapter 7. Evaluation of the methods and summary of user pilot is given in Chapter 8, and conclusions are given in Chapter 9.

2. REQUIREMENTS AND CONSTRAINTS FOR PROTOTYPE APPLICATION

TWIN has multiple requirements and constraints, resulting from the wanted behavior of the application, and the selected platform. The requirements and constraints presented in this chapter are divided into two categories: functional requirements for the user interfaces, and technical requirements caused by the target platform.

2.1. Requirements for user interfaces

Requirements for the user interfaces consist of general requirements for the functionality of the application, and more specific requirements for the features of the application. General requirements for TWIN are mainly: ease of use and scalability for a large number of users. Feature requirements are: ability to interact with others, ability to find who are around in the network and ability to share information and media with others.

2.1.1. Ease of use

TWIN should be easy and attractive to use. No user manual should be needed. There was no graphical designer in the project, but some user interface design paradigms were taken into account when designing TWIN. These paradigms were not considered as requirements, but as guidelines for the user interface design.

Most notably, no information should be hidden from the user. At all times, the possible actions are visible and in the same position on the right side of the screen. On the other hand, actions that can not be used are hidden from the user, preventing mistakes and decreasing the need for the application and the user to recover from those mistakes. [43]

2.1.2. Scalability for large number of users

As TWIN is designed to operate with users near each other, it should also function correctly when there is a large number of users near each other, such as in a sports or music event, or even in a lecture hall.

From the point of view of TWIN, this objective is mostly limited by the user interface. The interface must remain usable even with large number of peers. Amount and constant changes of the information presented in the interface must not obstruct the user.

2.1.3. Ability to interact with other users

As a social application, the purpose of TWIN is to allow users to interact with each other. Interaction is enabled by offering real time chat and long living textual messages.

2.1.4. Ability to find other users

To make interaction possible, users must be able to find each other. This is done by allowing users to create groups and to see other users in a global shared community.

2.1.5. Ability to share information and media

Sharing is an important aspect of social networking. Users must be able to share their personal information to other users, as well as so share media they have found or created.

2.2. Constraints of the platform

Target platform chosen for the prototype application is the Nokia N900 mobile computer. It was chosen, because it offered WLAN, Linux operating system and a touch screen. This platform sets some constraints for the application. While the touch screen is good for usability, it limits the layout of the interface. N900 also offers limited memory and computation time. The use of ad hoc WLAN sets constraints on the protocol and usage of bandwidth.

Figure 2.1. Nokia N900 mobile computer, showing the touch screen and a full sliding QWERTY keyboard.

2.2.1. Maemo platform on Nokia N900 mobile computer

Maemo [14] is an operating system for handheld computers. It was developed by Nokia, and it is based on Debian GNU/Linux [51]. The only devices using Maemo are the Nokia N series Internet tablets N710, N800, N810 and the latest N900. The development of Maemo has been discontinued, and the project was merged to MeeGo [24], a similar type Linux-based operating system for handheld computers and netbooks. Nokia N9 has been called a Maemo 6 device [48], but officially uses MeeGo 1.2 Harmattan as its operating system [16].

Maemo is a very flexible operating system. Majority of the applications that can run on a desktop Linux distribution can also run on Maemo. Virtually any programming language available for Linux can be used to develop software for Maemo. The only officially supported development environment is the GTK+ [57] library with the C programming language [15]. Tools for C++ and Python are funded by Nokia, but they are not officially supported languages [39, 20].

Most of the restrictions set by the target environment of TWIN are dictated by the N900 mobile computer, pictured in Figure 2.1. The N900 has a 600 MHz ARM CPU and 256 megabytes of memory. Its 3.5 inch display is very small when compared to desktop displays or tablets. On the other hand, the display resolution of 800x480 pixels is large when compared to the display area, resulting in a high pixels per inch (ppi) ratio of 267, comparable to the current high-end mobile computers. For

Table 2.1. Specifications of N900.

Feature	Value
Dimensions	110.9 × 59.8 × 18 mm
Weight	181 g
Display	3.5 inch touchscreen
Resolution	800x480 pixels (WVGA)
Permanent memory	32 GB Flash, extendable up 48 GB with a microSD memory card
Application memory	256 MB RAM + 768 MB virtual
Processor	TI OMAP 3430: ARM Cortex-A8 600 MHz,
Graphics support	PowerVR SGX with OpenGL ES 2.0 support
Keyboard	Full QWERTY keyboard
Network	GSM, GPRS, EDGE, WCMDA, HSPA, Bluetooth 2.1, WLAN b/g
Network security	WEP, WPA, WPA2
Camera	5 Mp autofocus, CMOS sensor, f/2.8-5.2
Video support	H.264, MPEG-4, Xvid, WMW, H.263
Container support	MP4, AVI, WMV, 3GP
Video streaming support	H.264, MPEG-4, Xvid, WMV, H.263 in AVI, MP4, WMV, ASF and 3GP containers
Music support	MP3, WMA, AAC, M4A, WAV

comparison, a 22 inch high definition display with a resolution of 1920x1080 pixels has the ratio of 100 ppi.

The N900 has extensive networking support, including GPRS [1], HSPA [50], Bluetooth [7] and WLAN [31]. While TWIN is network agnostic, it has been designed with ad hoc WLAN in mind. Maemo also supports multiple media formats and video streaming.

More extensive specifications of the N900 are shown in table 2.1. [17]

2.2.2. Ad hoc WLAN

Ad hoc is a mode of wireless LAN, in which the devices form the network with each other in a peer to peer manner, without the help of any external access points. With ad hoc WLAN it is possible to create high speed wireless links between two or more devices, regardless of the location.

The limitation to using ad hoc networks is, that every device must be in the range of all other devices it wants to connect with. The WLAN standard IEEE 802.11 does not define routing between the peers in ad hoc mode, but all communication must happen directly from peer to peer. To achieve communication between peers that

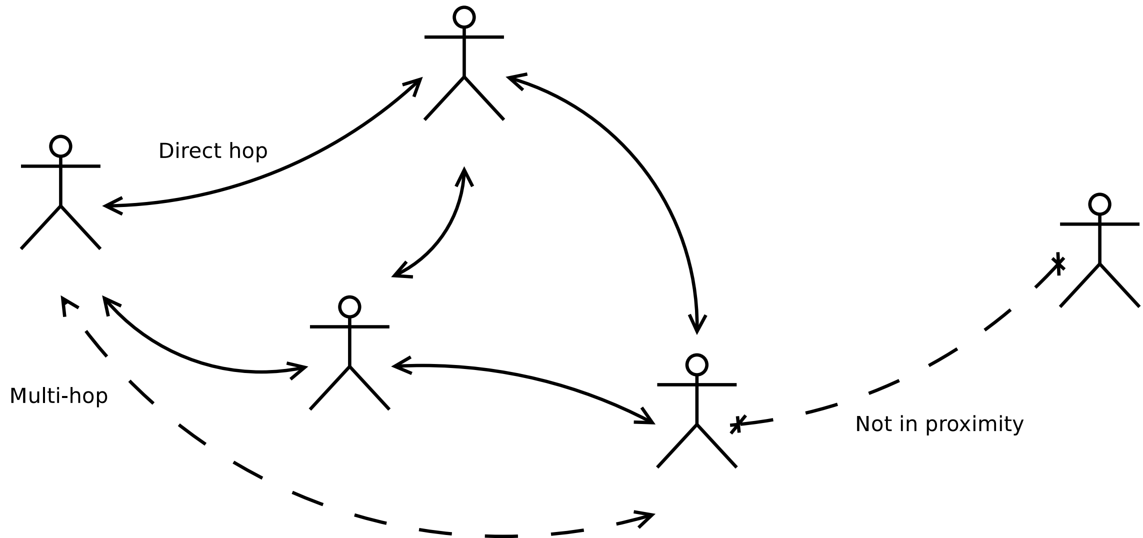

Figure 2.2. Example of a TWIN network. Users in proximity are directly connected with each other. Users too far for a direct connection may be available through multi-hop routing.

are not directly connected, a routing protocol that works on top of the data link layer is needed.

In the TWIN project, a separate protocol that routes traffic over multiple peers was used [2]. Routing over multiple peers is called *multi-hop routing*. Using multi-hop routing, the range of the ad hoc network can be extended to cover the area corresponding to the range of one peer device times the longest possible hop count offered by the routing protocol. A *hop* is the passing of traffic between two directly connected peers, thus the *hop count* being the number of hops in the chain of peers, when routing traffic between two indirectly connected peers. Figure 2.2. gives an example of the network.

When designing application for a battery operated mobile device, it is important to conserve energy. On network level, the multi-hop protocol enforced a 1 second transmission window in order to conserve energy. During the window, all queued packages were sent to and received from the network. The interval between transmission windows was 5 seconds. This way the radio could be turned off for 4 seconds, or 80% of the time. In addition the saving the energy used by the radio, this technique further reduces the number wake-ups for the CPU. The bandwidth of the network was also limited to maximize the range and reliability of the network.

Because of the limitations on the network traffic and bandwidth, limiting the number of packets sent to the network had to be taken in account also on the application level. The protocol was designed to minimize overhead, maximum amount of information was transferred in a single packet, and compression was used to fit more information

in packets. The network layer signaled TWIN for transmission windows, so no unnecessary network code was run, which led to fewer wake-ups for the CPU and thus to lower power consumption.

2.2.3. Scalability constraints from the platform

Memory and computation time available on N900 limit scalability. Other limitation is the network layer, which is something that TWIN can not fully control. WLAN has limited bandwidth, which will fill up when sending large amount of packets at the same time in the same physical area. This causes collisions between the packets.

WLAN takes this into account by using backoff times and retransmissions, but if there are too many collisions, no packets are transmitted successfully. TWIN addresses this problem by minimizing the amount of traffic, and by sending larger but fewer packets.

2.2.4. Touch screen interface

N900 has a touch screen, so the whole application should be controllable via tapping the screen. The small size but large pixels per inch value of the display sets requirements to what can be shown on the screen. The user interface elements must be large enough to be readable from the high ppi display, and to be tappable with even a large finger, but small enough to fit to the screen.

Dimensions of the screen are presented in Figure 2.3. Usable display area is actually only 800 × 424 px instead of 800 × 480 px because of the always visible bar located on the top of the screen.

2.2.5. User interface constraints

Maemo has user interface guidelines given on Fremantle Master Layout Guide [42]. These guidelines set requirements on how the user interface layout should be designed and what kind of user interface elements can be used. Elements sizes, application area, marginals, fonts, etc. are defined in the guidelines. The use of menus, tabs, toolbars and dialogs are also defined.

Windows always take up whole screen. Multiple open windows are stacked on top of each other, and moving between windows should be restricted to opening a new window and closing it by tapping the back button.

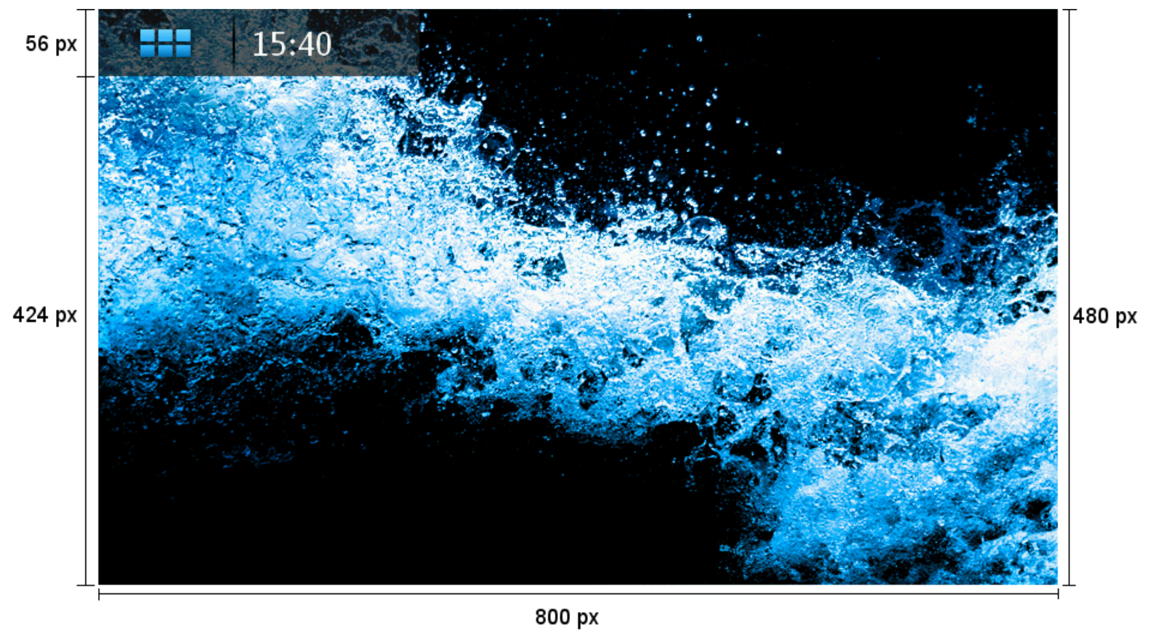

Figure 2.3. Maemo 5 home screen presenting the dimensions available for applications. The bar in the top of the screen is always visible, and takes up some of the display area from the applications.

2.3. Related work

There are many similar types of applications, with many notable differences. While all concentrate on discovering surrounding people and sharing personal information, the related works differ considerably from each other on their implementations. A summary of the related works in given in Table 2.2.

Serendipity [19], Wireless Rope [41], MobiSoft [36] and MobiClique [46] use Bluetooth as their network layer.

Serendipity, Wireless Rope, WhozThat [3] and Musubi [55] depend on central servers, and require an Internet connection to discover peers and exchange information.

Serendipity scans periodically for surrounding devices, collects their identifiers and sends them to a central server. The identifiers are associated with user profiles, which are then examined for a similarity score between people.

Wireless Rope also periodically scans for surrounding devices and uploads the encounters to a central server. Encounters are counted and grouped based on the number of encounters, or familiarity.

MobiSoft uses a combination of Bluetooth and ad hoc WLAN. It does not require central servers to operate. The devices automatically change information about

Table 2.2. Summary of related works.

Application	Network layer	Central servers	Reference
Serendipity	Bluetooth	Yes	[19]
Wireless Rope	Bluetooth	Yes	[41]
MobiSoft	Bluetooth	No	[36]
MobiClique	Bluetooth	No	[46]
WhozThat	Agnostic	Yes	[3]
Musubi	Internet	Yes	[55]

their users on encounters. The application evaluates the information and makes suggestions about similar people.

MobiClique also uses a combination of Bluetooth and ad hoc WLAN, and does not require central servers. The application implements a type of multi-hop networking by exchanging messages and passing the along to other devices.

WhozThat periodically advertises a unique identifier to the local network by some local area communication, e.g. Bluetooth. Identifiers are associated with social services on the Internet. This information is used to establish a social context of the surrounding people.

Musubi exchanges encrypted information of the users through a central server instead of local communication, in order to not reveal private information to the public. Users can form groups, send messages and share media.

Social networking on mobile environments has also been studied earlier in [34]. A paper describing design solutions in TWIN from a more technical point-of-view has been submitted for publication [37].

3. USE CASES FOR TWIN

As TWIN is a social application, the most important use case for TWIN is communication between people. Communication can be realized in several different ways, as described in the following use cases. Other important use cases are finding and following people, and sharing information with them.

These use cases described here are those which were implemented in the software. Other use cases were thought of, but they were not implemented because the ideas were seen uninteresting, and because of lack of time.

While developing the application, the specifications for the use cases have been very loose. Common example of a use case specification has been a few lines in an email, or few sentences over a phone call. Because TWIN is a prototype application, these ideas had to be quickly transformed into a viewable, or in the best case, usable demos. Quickly means that no formal specification could be formulated to follow the idea, but the development was done in an iterative process.

Many of the use cases have also been further developed during the pilot. One of the the main objectives of the pilot was to discover new use cases, and to develop improve the existing ones. These use case definitions are thus a product of the prototype, instead of a formal specification after which the prototype would have been developed. Figure 3.1. presents different use cases and their actors.

Following sections present the use cases, divided in to a rationale explaining the reasons behind the use case, description briefly explaining the use case, and scenario, giving a user story of an example situation, in which the use case might realize.

Each use case is also presented with an Unified Modeling Language (*UML*) activity diagram. In the diagrams, rounded rectangles present activities, diamonds present choices, rectangles present messages, and ellipses present references to other use cases. Arrows with regular heads present transitions between activities, and arrows with filled heads present message passing. Black circles present starting points and double circles present ending points. Concurrent actions of different actors are presented with swimlanes, separated with a single line. Interaction between actors is presented with a transition over the swimlanes.

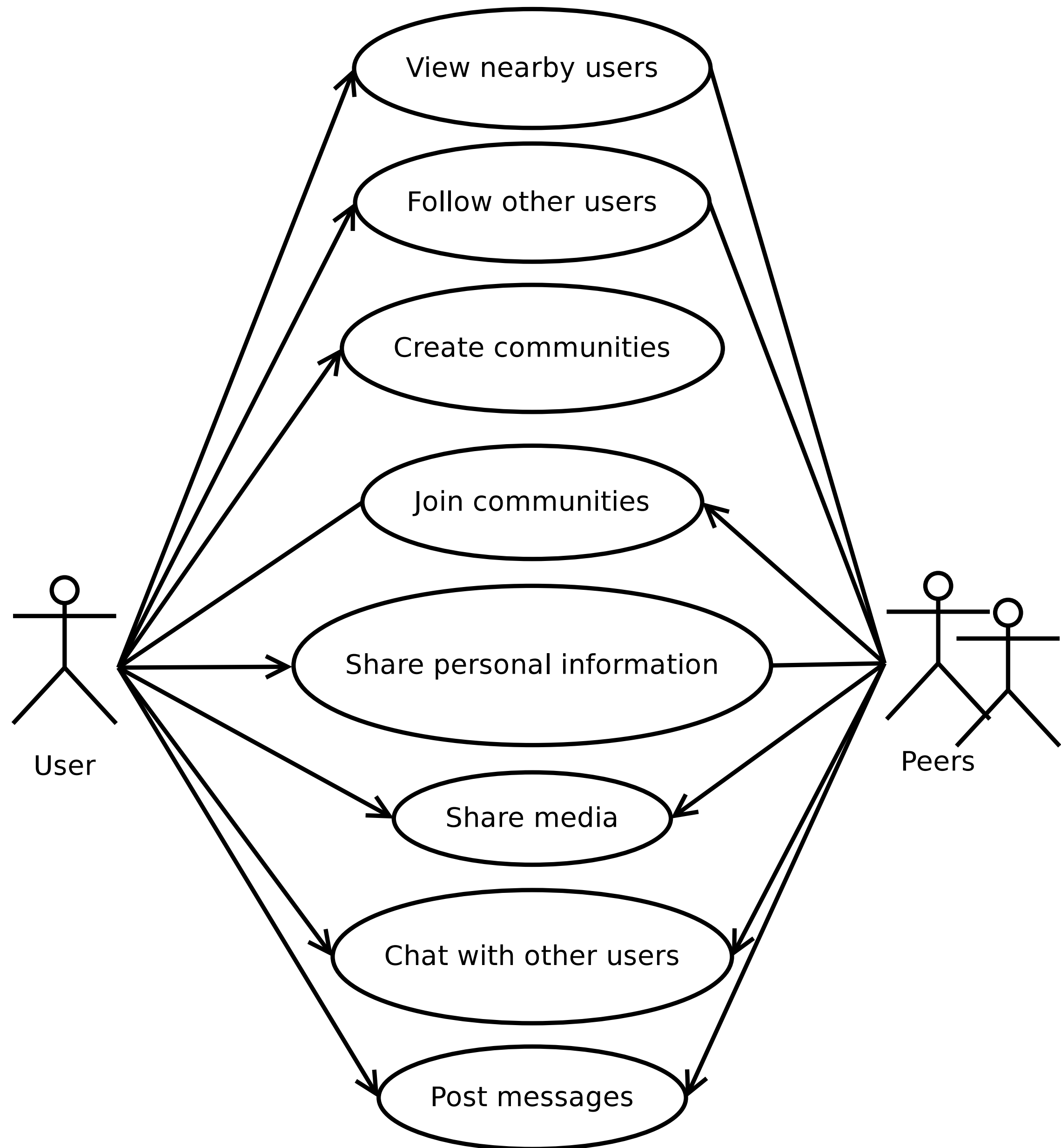

Figure 3.1. Use cases for TWIN. There are two actor groups: user being the one using the device and peers being people seen in the network. Arrow-headed connection indicates active participation to the use case, while a regular connection indicates passive role, meaning that no active actions are required from the actor, but the actor is a target of actions.

3.1. View nearby user

Rationale

Seeing who are near you at the moment and who belong to which communities is one of the most important features of TWIN, as it makes possible to use all other features.

Description

The user would like to see the people around him in a glance.

Actors

User

Peers (passive)

Scenario

The user has arrived at a location with new people, and he wants to quickly see if there are TWIN users around him. User opens the main view of TWIN and is presented with a list of those all communities with active peers. He opens the ”twin” community and sees all active users in the network.

User selects one of the shown peers, and is presented with a detailed view of that user’s personal information and communities in which she belongs to.

User then wants to see which peers are near enough so he can send files to them, so he opens a view showing nearby users divided according to their distance in the network. The user sees that one of the peers is close enough to receive a file, and proceeds to send her one.

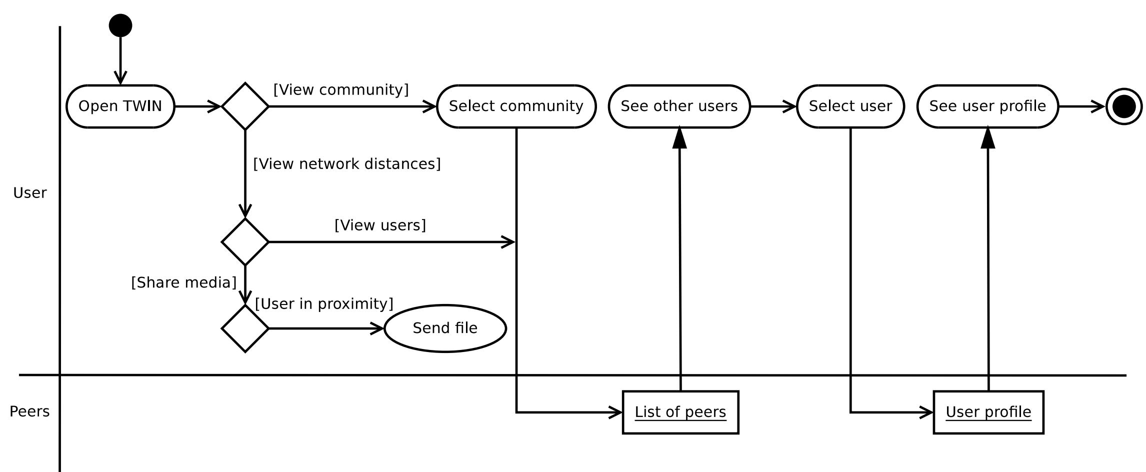

Figure 3.2. Activity diagram for viewing users use case.

3.2. Follow other users

Rationale

Users need a simple way of *a*) marking known users as friends, *b*) distinguishing friends from other users, and *c*) knowing when friends are around.

Description

The user has previously met other users, and wants to follow what they do and know when they come nearby.

Actors

User

Peers (Passive)

Scenario

The user finds out that he knows one of the active peers in the network personally. User marks her as his friend. He also sets a notification about friends joining in the settings.

Later, the same peer joins the network. User's device gives a notification, and user sees who has joined. There is also an icon distinguishing her as a friend from the other peers.

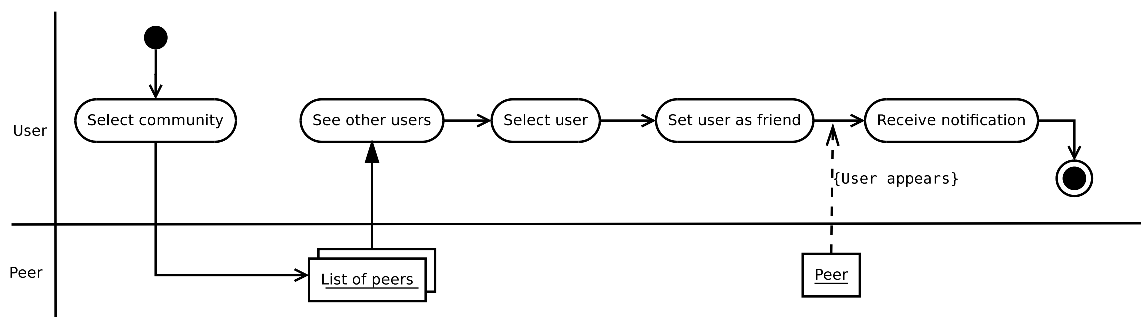

Figure 3.3. Activity diagram for following users use case.

3.3. Create communities

Rationale

People with similar interests naturally form communities. Communities help to find similar people, gather and filter content, or simply act as a temporary room for exchange of information. Communities can be public – which everyone can see and join to, or private – which are hidden and require a shared secret for the user to be able to join them.

Description

The user wants to create a community, to which other users may join. Communities can have a name, a description and an image to separate them from other communities.

Actors

User

Scenario

User is interested in frisbee golf and he wants to find other people interested in the same hobby. He creates a community by the name "frisbee golf" and selects to community to be public, so everyone can join it. User is automatically joined to the community. The new community now shows in the community view of TWIN and is advertised to other peers in the network by user's device.

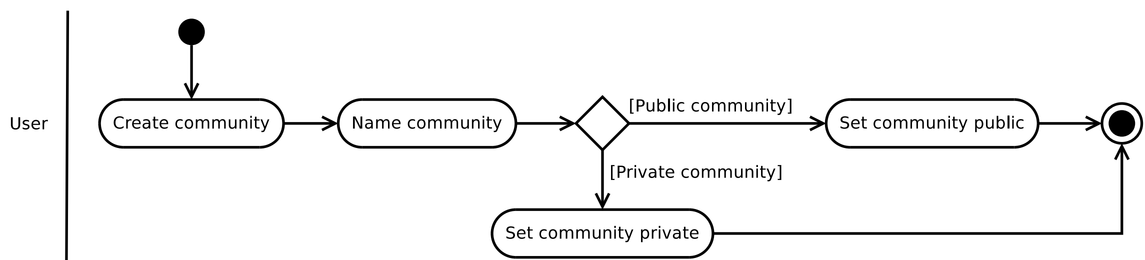

Figure 3.4. Activity diagram for creating community use case.

3.4. Join communities

Rationale

After a community has been created, users will need a simple way of finding and joining them.

Description

If a user wants to show interest in same group, she may join an earlier created community. After joining the community, the user will get announcements of people in the community joining the network or sharing media.

Actors

User (passive)

Peers

Scenario

User has created a community called ”frisbee golf”. A peer in the network is also interested in the sport and sees the community in her community view. She selects the community and joins it. Now she will receive messages and files published to that community, and can chat with the community members.

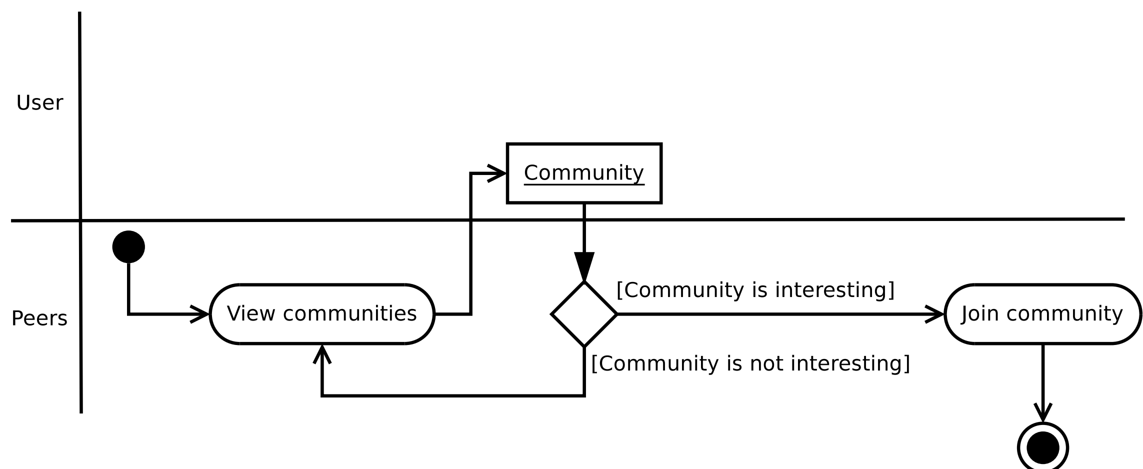

Figure 3.5. Activity diagram for joining communities use case.

3.5. Share personal information

Rationale

In a social network, it is natural to share information about one self. The information can be genuine or made up, and consist of any number of details. After joining the network, the user will want to see if there are other users nearby, and some additional information of other users, such as their profile images and nicknames.

Description

Users can add profile images and personal details, such as their real name, date of birth, address and languages they speak.

Actors

User
Peers (passive)

Scenario

User has joined the TWIN network and wants to share his personal information with other users. He opens his profile editor, where he can see different fields to fill in his information. He writes in his name, date of birth and hobbies. Then he taps on the default profile picture the change it. A dialog opens up, where user can select to use an image from the memory of the device, or to take a photo with the camera on the device. He selects to take a photo, and a dialog showing the front camera view is presented. User takes his image, and that image is automatically used as his new profile picture.

User wants to see if other users have shared their personal information. He opens a profile viewer for a peer and is presented with a formatted view of her shared information.

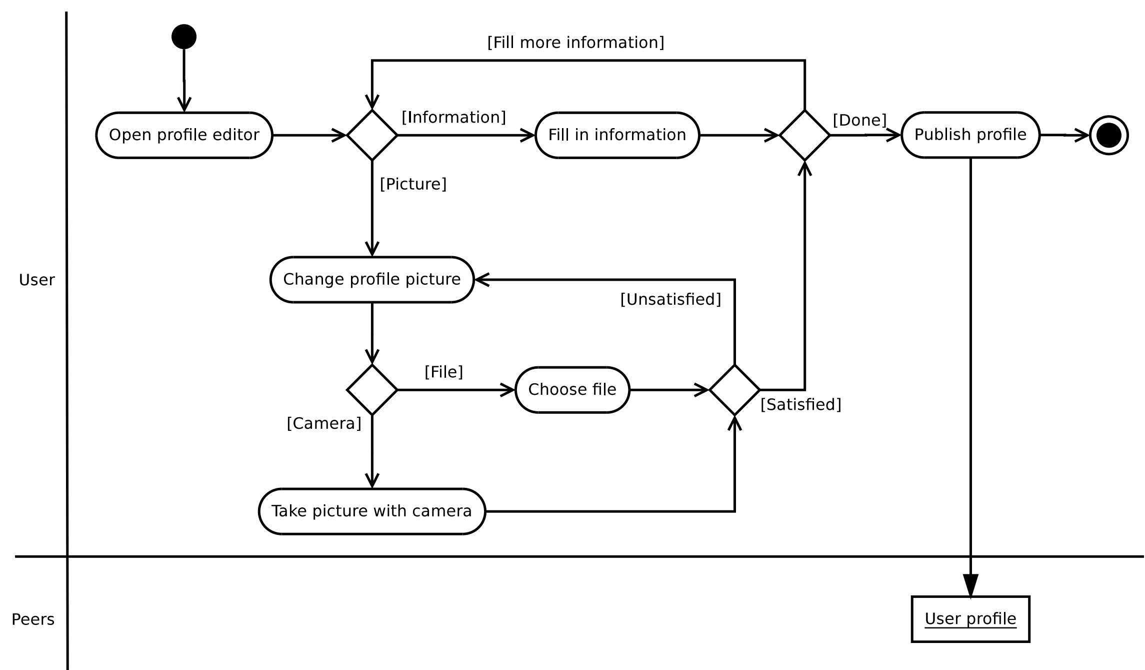

Figure 3.6. Activity diagram for sharing personal information use case.

3.6. Sharing media

Rationale

In addition to sharing personal information, users will want to share media.

Description

Files can be chosen from the device and shared to the network. The share can be public, or directed to a community or a single user. Files can be passively shared, or actively uploaded to other users.

Actors

User

Peers

Scenario

User has received a humorous picture, which he wants to share with his friends. He selects their private community and opens the file sharing dialog. He then opens the picture and selects ”upload”. The picture is sent directly to every device on the community.

User also wants other, unknown peers to be able to find the picture. He selects publish for the picture in ”twin” community and fills in keywords, so others can find it more easily. The picture is now available for everyone in the network.

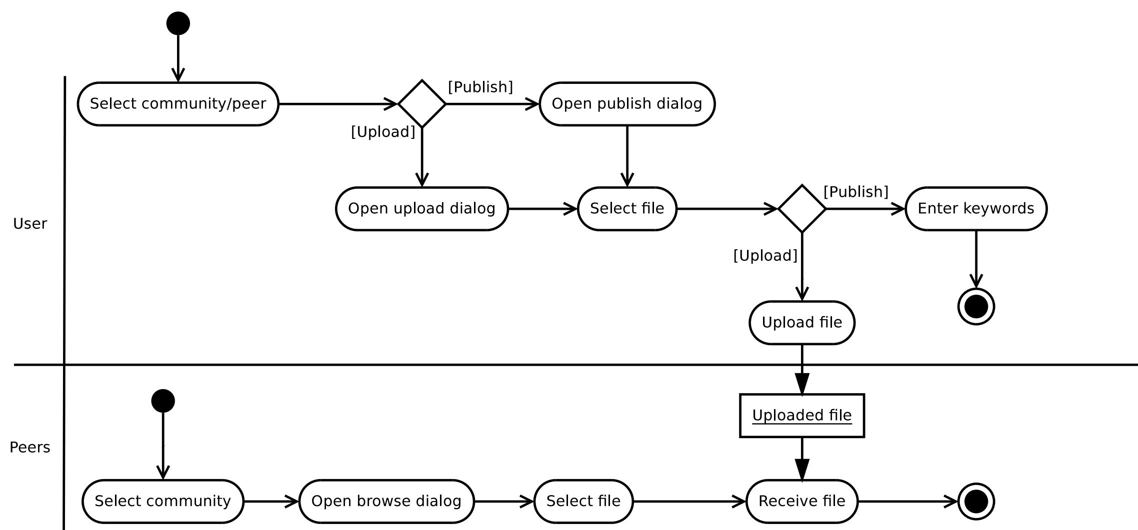

Figure 3.7. Activity diagram for sharing media use case.

3.7. Chat with other users

Rationale

Chatting, or sending short real time text messages, is the simplest way of communicating over the network. Chatting is also very popular among the users of different social networks.

Description

Users can chat publicly, between communities or between two individual users. Multiple chats can be in session at the same time.

Actors

User

Peers

Scenario

User wants to chat with his friend, who he sees is active in the network. He selects her user icon and starts a chat. A chat view opens with a tab for the new conversation. Other users in the network will not see the chat between the two peers.

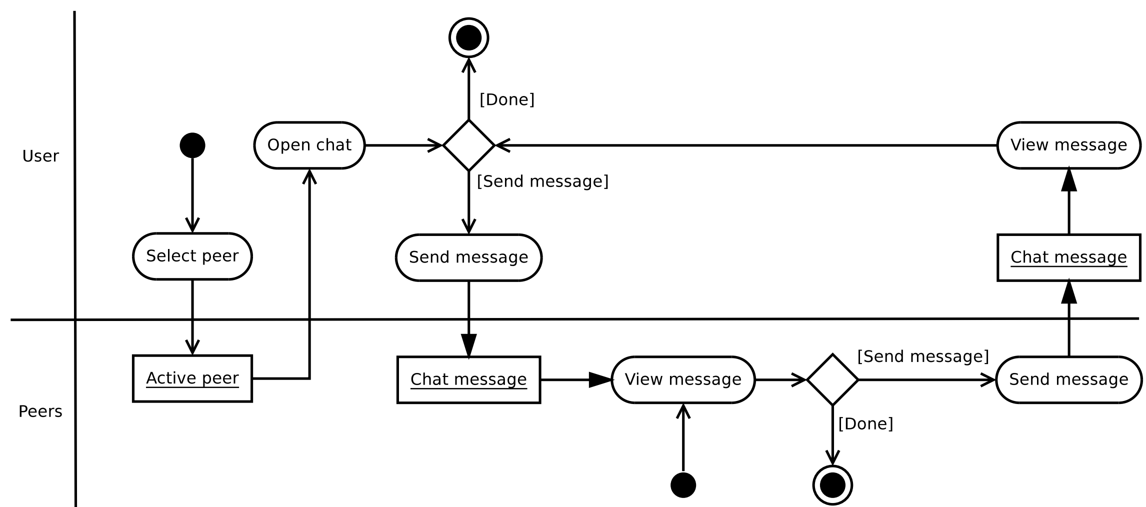

Figure 3.8. Activity diagram for chatting with other users use case.

3.8. Post messages

Rationale

Users should be able to post messages that have a longer life time, as opposed to chat messages.

Description

Message board is similar to sharing files. The messages are published and shared to other users. Messaged can also include files.

Actors

User

Peers

Scenario

User wants to sell his old phone after purchasing an N900. He selects the ”twin” community and opens the message board view. He writes a message and posts it to the community. All other active peers in the network receive a notification about a new message being posted.

One of the peers is searching for a used cell phone. She opens the message board and searches for messages about phones. She opens the message and sees the text posted earlier. As the phone seems interesting, she opens a chat with the user to ask for more details.

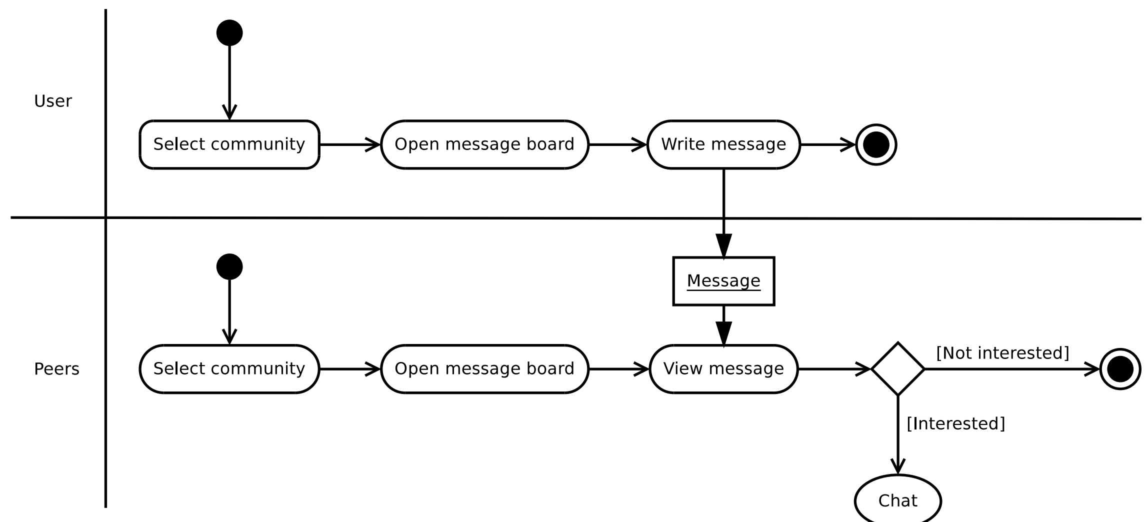

Figure 3.9. Activity diagram for posting messages use case.

3.9. Expressing mood

Rationale

Displaying mood graphically gives peers an easy way of finding out what is on other users' minds.

Description

Users can set their mood with an icon and a status text. The icon will be shown on their user icon.

Actors

User

Scenario

User feels bored and wants someone to cheer him up. He opens the mood changing dialog and is presented with a list of different mood icons. He selects bored and writes a status text asking for someone to cheer him. The mood icon will show to other peers on the user's icon.

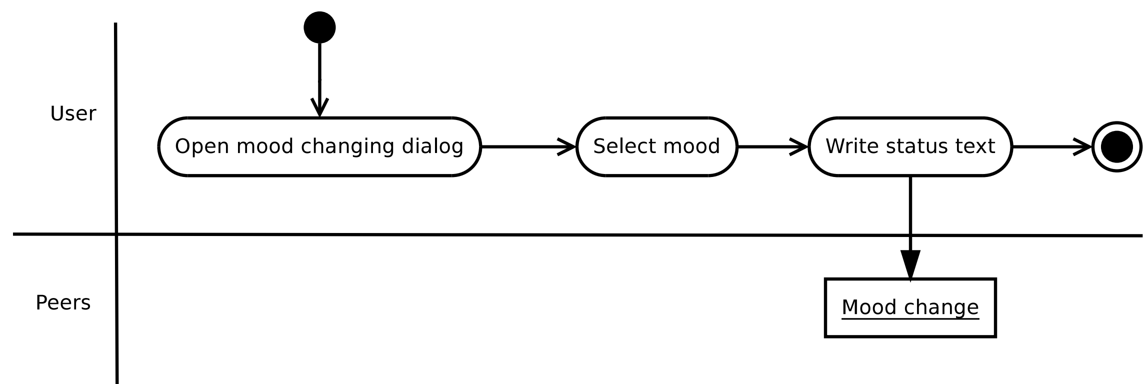

Figure 3.10. Activity diagram for expressing mood use case.

4. ARCHITECTURE OF TWIN

TWIN is divided into plugins to separate different functionality into distinct parts. By using plugins, it is easy to add new features without changing the core functionality of the program.

4.1. Plugin interface

TWIN core offers a uniform plugin interface, through which the core of the application and plugins can communicate. Figure 4.1. visualizes this technique. The interface consists of functions used to register and initialize new plugins, retrieve instances of registered plugins, and use services offered by these plugins. The services selected were those seen to be the most often used. In addition to the main plugin interface, plugins offer their own interfaces for other plugins to use.

Plugins receive messages using a *publish–subscribe pattern*. Publish–subscribe [5] (also called *observer pattern* or *signals and slots*) is a programming pattern where the listening part subscribes to only the data it is interested in, and the publishing part sends the data only to the subscribed parties.

There are several different ways of implementing this pattern (e.g. Qt's signals and slots mechanism [6]), but in TWIN it was implemented as a simple callback mapping, where the plugins subscribe to data by registering a function, and core then calls the registered functions when data is available.

4.2. Core / UI separation

As a result of the plugin architecture, it was simple to separate core functionality and user interfaces (*UI*). This allowed to develop core and interfaces separately, and to make changes in them without affecting others. This in turn allowed the development tasks to be efficiently divided to different developers. Especially user interface changes were simple to make, as they did not involve changes to core functionality, but only to the appearance and features of the user interface. The user interfaces were also simpler to develop, as common functionality between different

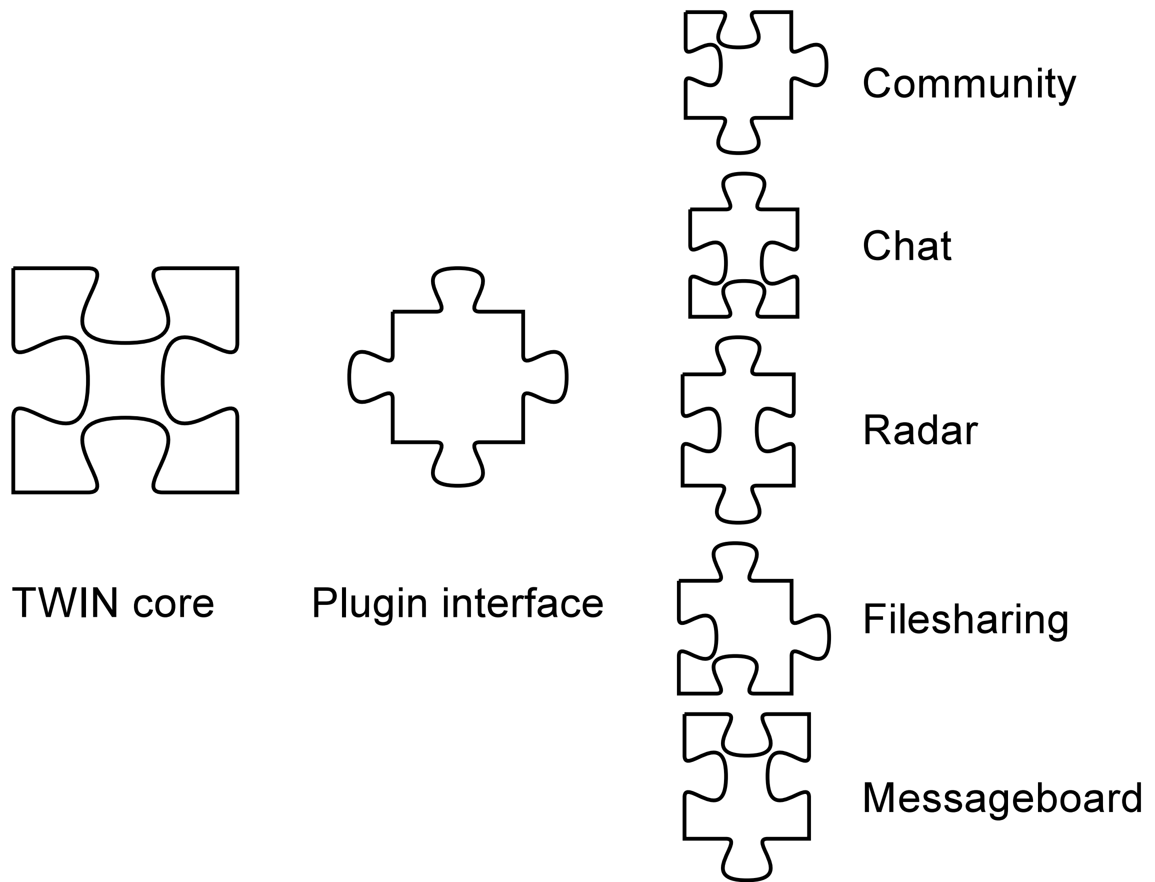

Figure 4.1. TWIN core connects to plugins through a uniform plugin interface to allow easy development of new features while being independent of other plugins and changes to the core.

user interfaces could be moved to a lower lever plugin, so that the code needed for that functionality was written only once, and then exposed to use of other plugins through the plugin interface.

Although core functionality and user interfaces are separated, they depend on each other through the plugin interface. When developing the application without specifications, the interface tended to change, sometimes very rapidly and dramatically. This caused some problems with the core / user interface interaction.

The separation was not fully complete. Because of imperfect interfaces, user interface plugins had to keep track of some of the information that was retrieved from the application core. Thus the solution was not a pure implementation of *model–view–controller pattern* (MVC) [8], which is a design pattern to separate user interface from the core functionality.

In MVC pattern, model would be the core – the representation of all of the data and state information in the application; view would be the graphical presentation of the

state through the user interface; and controller would be parts of the user interface that take input from the user and send the back the model.

In TWIN, this separation is partly overlapping, as the user interfaces contain some of the functionality reserved for model. Management of user lists in the user interface code is the most notable example of this.

4.3. Architecture

The modular plugin system of TWIN enables the application to have only few dependencies between modules. Figure 4.2. presents the architecture design of TWIN as a stack diagram. Modules in the upper layers of the diagram use services from the lower level modules. There are also dependencies between the modules in the same layer, but no lower level module has dependencies on any upper level module. Dependencies between plugins have been kept one-directional to avoid cross-dependencies. If common functionality was found, it was moved away to lower levels.

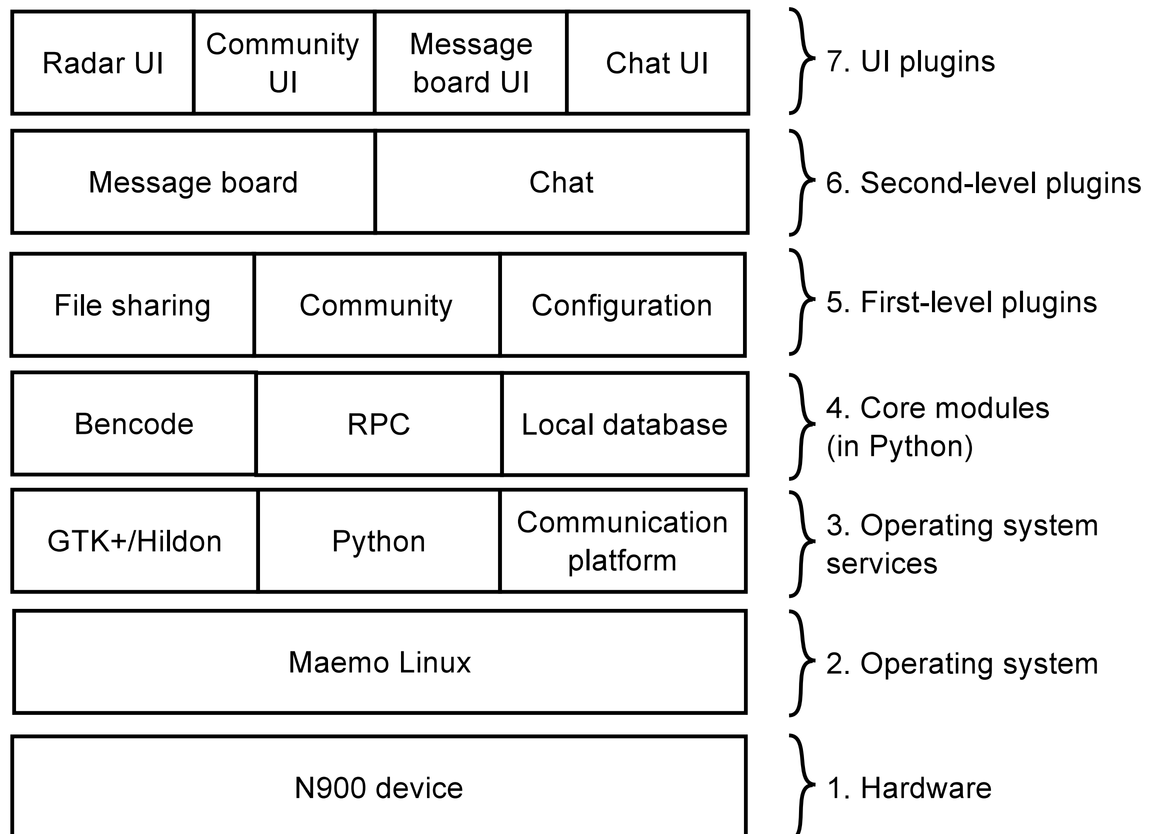

Figure 4.2. Architecture design of TWIN as a stack diagram. Upper layers use the services from the lower layers. This diagram does not include small utility plugins that are not part of the main functionality of TWIN.

Modules developed for TWIN are on the layers 4–7 of the diagram. On layer 4 are the core modules bencode, RPC (*remote procedure call*), and the local database. RPC module uses the communication platform to send messages over the network. These messages are encoded with bencode [13], which is a light-weight serialization format for basic Python data structures. Local database module offers services for serializing data to the persistent memory on the device.

On layer 5 are first level plugins file sharing, community and configuration. File sharing implements functionality needed to share and download files. This functionality is implemented using the RPC layer with bencode. Community holds the data about users and the state of the network, and stores its data to the local database. Configuration offers an interface for application-wide settings database.

On layer 6 are second level plugins message board and chat. Message board uses file sharing plugin to share messages. Chat is implemented as a separate plugin, but uses services of bencode and RPC layer.

On layer 7 are user interface plugins for radar, community, message board and chat. Radar plugin consists of only a user interface with no state information of its own. It visualizes the state of community plugin. Community user interface plugin offers the main user interface from TWIN. Message board and chat user interface plugins implement interfaces to their respective second level plugins.

5. IMPLEMENTATION

This chapter presents the methods and tools that were used to implement TWIN. Implementation of the plugin interface is also presented. Tools and languages presented here were selected mainly on the principle of being easy, and thus fast to use, to develop prototype software.

5.1. Methods

No formal agile methods, such as Scrum [49], Extreme Programming [4] or Feature-driven development [45] were followed during the development.

Scrum uses an iterative method for project management. It enforces roles, such as *Scrum Master*, who is responsible for the deliverables of the team. Scrum uses *sprints*, which are time-restricted cycles of development with fixed goals.

Extreme Programming also uses time-restricted development cycles. It enforces programming in pairs and unit testing of all written code among other things.

Feature-driven development requires features lists to be built before the development is started. These features are then split into small groups, which are developed in time-restricted cycles.

These methods would not have been suitable to use in the TWIN project, because the developers had no previous experience in them, and because the methods would not have fit to the irregular working hours in the project. All of them require timely cycles and fixed deliverables, which do not fit to the principle of prototyping ideas. Feature-driven development requires that the features are planned before the development starts, which does not fit the goal of recognizing new ideas during development.

Instead, some custom methods were agreed to help the development flow: Every change should be tested by the developer who wrote the code before making a commit; commits should be tagged to ease code review; every commit should be reviewed before merging them into developers own branch; every commit should

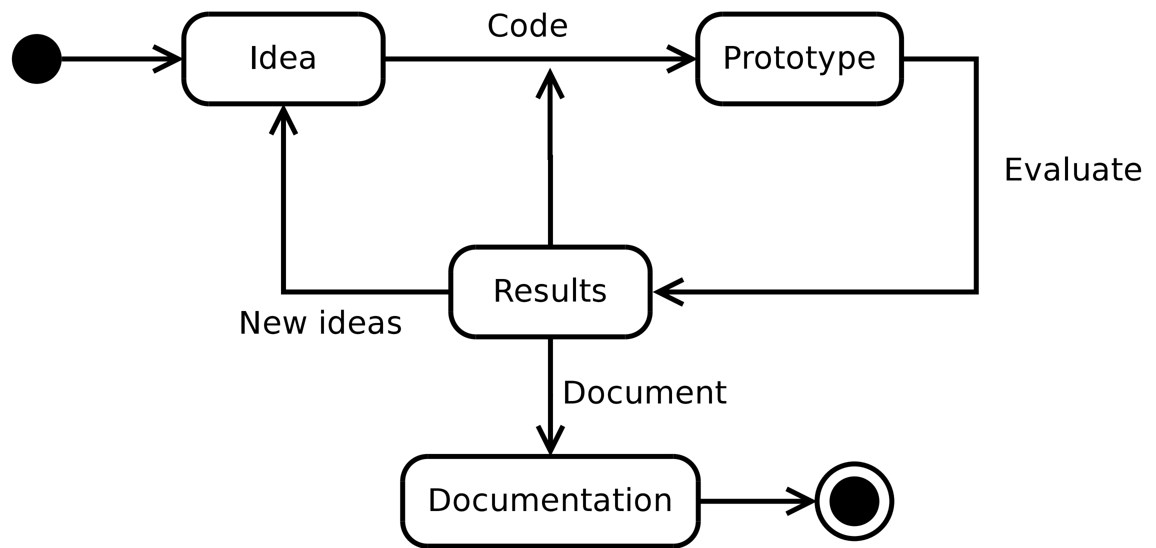

Figure 5.1. Development flow of TWIN. Ideas are made into prototypes, which are then evaluated. This may lead into new ideas or new code. Finally, results are documented.

leave the source tree into a usable state; and fast iteration of features should be preferred over completeness of the commits.

An outline of the development flow is presented in Figure 5.1. First, the idea is formulated. Then the first prototype code is written, and it is evaluated for the implementation of the idea, code quality and possible bugs. This evaluation may lead into new ideas, or needs to improve the code. New code is then again evaluated.

5.1.1. Fast iterations

As the requirements and features of TWIN were constantly changing along with the progress of the research, it was important to be able to prototype ideas fast. To make this possible, no specifications were written in any point. New features were tried by implementing them and evaluating the outcome. Bad design decisions were simply reverted, and good decisions were developed further.

To be able to experiment with new features as fast as possible, fast iteration of features was needed. This required many small changes to the code instead of few large ones. This way the possibly needed changes could be done sooner.

We also used a bottom-up approach on new features: first, the most basic functionality is added and tested, and after that the feature is iterated to add more functionality, polished looks and cleaner code.

5.1.2. Immediate testing

With no specifications for the prototype, it was hard to verify that the implementation is valid. To mitigate this problem, it was agreed that the developer who writes a feature will first test it and, only after being convinced of its functionality, commit it to the version tree.

Because the writer of the code is in many times blind to one's own mistakes, it was agreed that before other developers merge code to their own branches, the code must be reviewed. This way all code will be reviewed by every developer.

If a bug is found, usually the founder would report it to the writer of the code. But, if the writer is not present, the founder is free to fix the bug. Thus the code is not owned by its writer, contrary to what is customary in heavily managed projects.

While fast prototyping of new features was important, keeping the code functional was even more so. If a bug was found, it was crucial to first fix the bug, and only then carry on writing new code.

5.1.3. Tagging commits

To help reviewing the code, every commit was tagged according to its intended effect. Nine different tags were used: perfective, corrective, cleanup, preventive, documentation, security, adaptive, workaround, and breakage. These tags follow those described in [38], with the addition of some custom ones to better describe the commit. The usage of these tags in TWIN are explained in Table 5.1.

Table 5.1. Tags used to mark different commit types, not accounting for merge commits.

Commit type	Description	# of commits	% of commits
Perfective	Adding or improving functionality	1195	43.0
Corrective	Fixing incorrect functionality	759	27.3
Cleanup	Improving quality of code without changing functionality	302	10.9
Preventive	Preventing incorrect functionality	168	6.1
Documentation	Only adding missing comments, no changes to functionality	27	1.0
Security	Preventing security holes	17	0.6
Adaptive	Adding support for different target platforms	15	0.5
Workaround	Temporary or low-quality solution to a problem due to problems presented from elsewhere in the code or 3rd party libraries	12	0.4
Breakage	Workaround that breaks some other functionality	1	<0.1
# of tagged commits		2496	89.9
# of commits		2775	100

5.2. Tools

Multiple tools were needed to implement TWIN. These tools consist of programming languages and interpreters, libraries and a version control system.

Most of the tools were readily available to reuse. We also needed few custom tools, for assuring the quality of the code and to make trying out changes easier. These tools were written in Perl [12] and Bourne Shell [40] languages for their suitability for using other programs as a part of a new program and easy access to regular expressions. Table 5.2. lists the most important tools used.

Adhoc-up tool was needed to quickly set up ad hoc network on the device. It deconfigures the current network and sets up an ad hoc network with the SSID ”twin”. It removes the need to set up network parameter by hand, which is time consuming and error prone.

Find-unused-imports, find-unused, and find-unused-python-functions are scripts to maintain code quality by removing unused code, which could lead into bugs. They search for unused imports, modules and methods inside modules, respectively.

Make-release tool creates an encrypted package of the source code, while upload tool uploads the code to the device. The purpose of these tools is simply to automate often repeating tasks and to cut down development time.

Table 5.2. Custom tools used in development of TWIN.

Tool	Description	Language
adhoc-up	Setting up ad hoc network on N900	sh
find-unused-imports	Finding unused Python imports in modules	perl
find-unused	Finding unused modules	perl
find-unused-python-functions	Finding unused Python methods in modules	sh
make-release	Packaging current version to an encrypted file	sh
upload	Uploading the source files to an N900 device	sh

5.2.1. Git

The version control system used when developing TWIN was Git [10]. It was chosen based on two criteria: 1) it was familiar to the developers; and, more importantly 2) it is distributed.

Distributed version control means that the version control system does not dictate a central master repository to which every developer must commit their changes, but instead every repository features its own branch of the source tree, and holds a full version history for the project.

A branch is a distinct copy of the source tree, which differs in some way (at least by its name) from other source trees. Branches can be modified in parallel, and their changes can be merged together to either form a single branch with changes from the merged branches, or to just keep the branches synchronized.

Because of this distributed nature of Git, every developer is able to make independent changes to their working tree. Unlike with centralized version control systems, developers do not have to fetch changes committed by other developers before committing their own changes. This way new features can be developed without considering changes made by others. After the feature is completed and tested, it can be merged by other developers. This gives a great deal of freedom in development.

5.2.2. Python

The Python programming language [23] was used to write all of the main program and plugins. Python is a dynamically typed, interpreted language.

Dynamic typing means that there are no compile-time types checks, but the types are checked when the variables are used in run-time. In addition, Python does not enforce variables to be certain types before they can be used. Instead, if variable must merely hold the member it was asked for in order to be used. This is called *duck typing* [9] in Python.

Python features a very extensive standard library. The library includes modules to interact with e.g. the operating system, file system and network. It also includes basic data structures, such as lists and hash tables, or key-value dictionaries as the structure is called in Python. Table 5.3. lists the most important Python library modules used, and TWIN modules using them.

In addition to Python's standard library [28], five different Python libraries were used: PyGTK [47], python-dbus [26], python-hildon [32], python-gst [29], and

Table 5.3. Most used Python Standard Library modules in TWIN.

Module	Description	Used by
StringIO	In-memory file like object	twinawarenet, twinconfigparser
copy	Methods to deep copy Python objects	filesharing, meta, twinstate
datetime	Date and time related classes	scheduler, statistics, utils
dbus	Interprocess messaging bus	vibra, wlancontrol
errno	Standard error symbols	community, filesharing, ioutils, listener, ossupport, statistics, twinawarenet
fcntl	File descriptor control	ioutils
mimetypes	MIME guessing	filesharing_gui, openfile
os	System dependent OS routines	community_gui, content, feedback, filesharing, filesharing_gui, filetransfergui, gui_user, guihandler, guiutils, ioutils, keymanagement, keymanagement_gui, messageboard_gui, messaging_gui, notification_gui, openfile, options, ossupport, ossupport, pathname, pic_choose_dlg, radar, scheduler, sendfile, sendfile_gui, simple_tracer, splash, statistics, twinawarenet, twinawarenet_gui, twinconfigparser, twinstate, watches_gui
random	Random number generation	community, fetcher, filesharing, keymanagement, main, messageboard, messaging, tcpfetcher, twinawarenet, utils
shutil	File copying methods	ossupport
signal	POSIX signal support	ossupport
socket	Socket support	community, ioutils, listener
struct	Conversion methods between C and Python data structures	ioutils
subprocess	Process spawning and I/O	openfile, ossupport
sys	Python interpreter related methods	main, options, simple_tracer, support, twinawarenet, twinstate, twintracer
time	Time manipulation	community_gui, communitymeta, feedback, messageboard, messageboard_gui, messaging, scheduler, statistics, user, utils,
zlib	Fast data compression	twinawarenet, utils

Table 5.4. Summary of used 3rd party libraries.

Library	Purpose	Reference
PyGTK	Graphical user interface library	[47]
python-dbus	Message bus system	[26]
python-hildon	Application framework for Maemo	[32]
python-gst	Multimedia framework	[29]
bencode	Data encoding	[44]

bencode [13]. Table 5.4. describes these libraries. Python-dbus library is for using the D-Bus [25] inter-process communication (*IPC*) library. It is mainly used to control the peripherals on the device, such as the vibra. Python-hildon contains Maemo specific user interface widgets. Python-gst is a binding for GStreamer [53], a library for handling media, for example playing music or video on the device. Bencode is a serialization library. PyGTK is discussed in the next sections.

5.2.3. GTK+

GTK+ is a user interface toolkit, originally designed for the X11 desktop environment for use in Linux-based operating systems. It includes user interface widgets to build interfaces from reusable pieces, such as windows, buttons, text fields, images, tabbed interfaces, and containers to organize the widgets.

GTK+ also includes a lower level utility library called GLib. It includes several data structures missing from C, such as dynamic arrays and binary trees. As we used Python, which implements these basic tools, they were not used in the prototype development. GLib also offers tools for timers, function callbacks and waiting for input/output events from the operating system. These tools were used in TWIN. [57]

5.2.4. Portability of Python and GTK+

A great benefit from using Python and GTK+ was their portability between different platforms. Both are available for x86 PC architecture, as well as for the ARM architecture used in N900.

This allowed us to develop and test the prototype on a regular PC. Only after a feature was found to be ready, it was tested on the target device. As Python is an interpreted language, no actions were needed other than transferring the program to the device.

N900 has limited calculation capacity when compared to an average desktop PC, and executing the program on the device has a noticeable delay of around 20 seconds. Thus the possibility to test to prototype on a fast PC saved a great deal of development time. The delay is no issue when using the program, as it is designed to be always on.

5.3. Plugins

Plugin interface offers the plugins methods to register services they offer to other plugins, use services offered by other plugins and subscribe to data coming from the core. To the application core the interface offers methods to initialize and cleanup plugins, and to publish data to them.

The following sections present the main methods of the plugin interface.

5.3.1. register_plugin

Method `register_plugin` allows a plugin to register itself for use of other plugins. Plugins are recognized simply by their type. Other plugins can then use services from registered plugins by retrieving their instance using `get_plugin_by_type` method.

5.3.2. get_plugin_by_type

This method retrieves a plugin registered earlier with the `register_plugin` method. It returns an initialized instance of the retrieved plugin. Plugins are *singleton* objects, so every call to this method will return the same instance.

5.3.3. ready

This method is called by the core after every non-UI plugin has been initialized. The purpose of this method is to offer a place for the plugins to initialize services needed from other plugins. If services would be initialized before the plugins were ready, some plugins might be in an undefined state.

5.3.4. gui_init

The `ready` method was called after initialization of non-UI plugins. This `gui_init` method is then called by the core to initialize UI plugins. The purpose of this method is to ensure that the state of the plugin is finalized, including all of the services used from other plugins, before any information is presented to the user.

5.3.5. cleanup

Method `cleanup` is called by the core when TWIN is terminated. In this method, plugins should save any persistent data, close open network connections and free any reserved resources.

5.3.6. user_appears, user_disappears, user_changes

There methods are called by the core when a user appears, disappears, or when the state of a user changes, respectively. These methods publish information about the network to the plugins.

5.3.7. community_changes

This method is called by the core when community information is changes. This means a change in communities profile or icon.

5.3.8. Example plugin

The simplest possible plugin for TWIN could be implemented in 11 physical lines of code, including white space for good readability. Example of this is shown in Programme 5.1. The simplest plugin that was actually implemented, vibration, is shown in Programme 5.2. Vibration plugin alerts the user by vibrating the device, when a friend joins the network or a file is received.

First, core calls `init` function of the plugin module, which creates the plugin object. Python then calls the initializer method of the plugin object, `__init__`. In this method, the vibra of the device is initialized, and the plugin is registered, so other plugins may use services offered by it, mainly the method `vibrate`. Then `ready` method is called by the core. There the plugin registers a callback function to sendfile plugin. This callback function, `file_receive`, simply calls `vibrate` method of the plugin the vibrate the device and alert the user. Another method that calls `vibrate` is the `user_appears` method, which is part of the plugin interface. When a user appears in the network, this method is called by the core. If the appeared user has been marked as a friend, `vibrate` is again called to alert the user.

Programme 5.1. Simplest possible plugin.

```
1  from plugins import Plugin
2
3  class NewPlugin(Plugin):
4      def __init__(self):
5          self.register_plugin('new-plugin-name')
6
7      def user_appears(self, user):
8          print '{nick} appears!'.format(nick=user.get('nick'))
9
10 def init(options):
11     NewPlugin()
```

Programme 5.2. Simplified example of plugin that vibrates the device, when a friend joins the network or a while is received.

```
1  class Vibra_Plugin(Plugin):
2      def __init__(self):
3          # code to initialize phone vibra
4          # ...
5          self.register_plugin(PLUGIN_TYPE_VIBRA)
6
7      def ready(self):
8          sendfile = get_plugin_by_type(PLUGIN_TYPE_SEND_FILE)
9          sendfile.receive_cb.append(self.file_receive)
10
11     def get_vibra_enabled(self, profile):
12         # code to check phone-wide vibra settings
13         # ...
14
15     def profile_changed_handler(self, foo, bar, profile, *args):
16         self.get_vibra_enabled(profile)
17
18     def vibrate(self):
19         if self.enabled:
20             self.mce.req_vibrator_pattern_activate(
21                 'PatternChatAndEmail',
22                 dbus_interface='com.nokia.mce.request')
23
24     def file_receive(self, cb, user, fname):
25         self.vibrate()
26
27     def user_appears(self, user):
28         if user.get('friend'):
29             self.vibrate()
30
31 def init(options):
32     Vibra_Plugin()
```

6. USER INTERFACE PROTOTYPES

User interfaces were specifically designed to be used with the touch screen of the N900. This required the UI elements to be large enough to be used with a finger. On the other hand, the small size of the display set limit to how much information can be displayed at the time.

Because of limited display area, some UI elements had to be hidden to menus and tab pages. This is against the Maemo user interface guidelines, but was seen as a necessary action to be able to accommodate all of the actions in the application. Actions hidden were the ones that were most seldom needed: settings, plugin actions and additional information. Core functionality is always visible on the views.

6.1. Community view

Community view shows all those communities that have at least one active user at that time – including the user of the application. Example of the community view is shown in Figure 6.1. On the upper left corner is always the main community named "twin". It is a special community to which all the peers in the network belong to, and its default community icon differs from other communities. Like other communities, that icon can be changed, but the name of the community can not, and the user can not leave the twin community. After the twin community come all the other communities.

On the right side are the community action buttons. A community can be selected by tapping it once. Then an action can be performed on the community by tapping one of the action buttons.

Community button lets the user create new communities and join the existing ones. Chat button opens a chat to the selected community. Filesharing button lets the user publish and search files from the community and msg board button lets the user read and write persisting messages. Text on the message board button had to be shortened because of limited display space.

After opening a community, a list of active users is shown. User listing of "twin" community is shown in Figure 6.2. The action bar on the right is modal, and now the

Figure 6.1. Community view showing twin community, which is common for all users, and two other communities, Tampere and Hervanta.

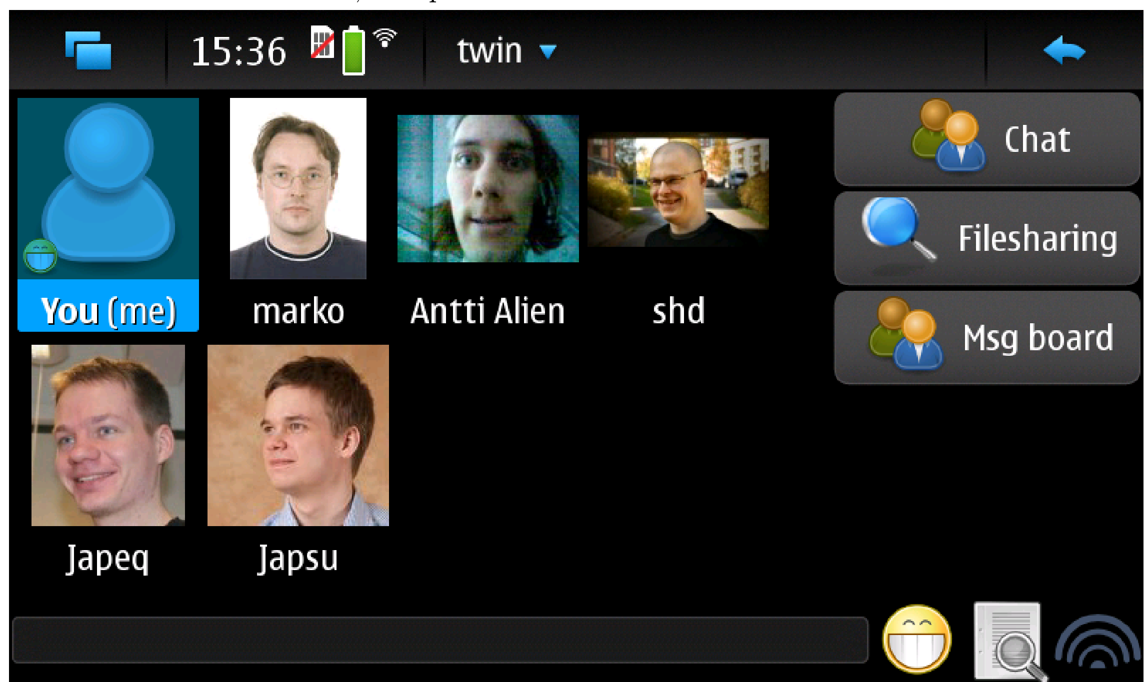

Figure 6.2. View inside the twin community, showing author with the nickname Antti Alien, and other members of the research team.

actions are performed to users, not communities. User's own profile icon is always shown on the upper left corner, with a bold formatted text, prepended with "(me)". All other users are added to the view in the order of appearance to the network.

As users joined and left the network, it was noticed that the user list is very restless and hard to use when icons change their places. This was solved by making the list stable: when a user leaves the network, user icons do not change place, but there will be a hole in the place of the user who left.

Community view holds its own version of the user and community databases, which must be kept synchronized with the core. This is done by using the `user_appears`, `user_disappears`, `user_changes` and `community_changes` methods from the plugin interface. Users are added and removed, and their information is changed in these methods. Communities are drawn according to the number of users they have: after the last user (accounting the local user) has left the community, it is no longer drawn in the community view. Only references to users and communities are held in the database. All additional information is retrieved from the community plugin when it is needed.

6.2. Profile view

By tapping a user, their profile is shown. Example of a profile view is shown in Figure 6.3. The profile is formatted from the personal data shared by the user. Again

Figure 6.3. Personal profile of user with the nickname Smith.

Figure 6.4. Profile editor for sharing personal information.

the action bar is shown on the right. The ”More actions” tab contains additional actions, and actions added by plugins. From the tab, user can be invited to a community and the profile information can be refreshed. The ”User communities” tab contains a list of all of the communities the user is in.

Profile editor view allows the user to share personal data. Possible fields include name, age, gender, place of residence and a free form description. Nickname used in the network along with the profile picture can also be changed from the profile editor view. All fields are freely formatted, allowing the user to input any data in them. Profile editor is shown in Figure 6.4.

These dialogs hold no information of their own per se, but they still must be synchronized with the community plugin. This is because there is no interface to fetch possible profile fields from the community plugin. This again is because the field names displayed to the user are user interface specific information. Possible fields and their names must therefore be upheld by the developer in order for them to be displayed. Information to fill these fields is retrieved using an interface offered by the community plugin.

6.3. Radar view

Radar view was the most popular user interface feature on TWIN in the pilot [56]. It allows users to easily see peers who are nearby. The view is divided into circle

sectors, which indicate the peers' distances in the network from the user in terms of *hop count*. Radar view visualizing hop counts on "twin" community is shown in Figure 6.5.

First circle indicates that the hop count is 1, meaning that there is a direct connection between the user and the peer. Second circle indicates that the connection is established through one of the peers in the first circle, and the third circle indicates that the user is further than 2 hops away.

Because of how the underlying network works, these distances roughly equal real world distances. Peers on the first circle are in most cases closer to those in outer circles. There were occasional exceptions to this, likely resulting from the network layer dividing work required for routing. Walls and other objects blocking and reflecting the wireless signal also affects the outcome.

On the other hand, the view does not distinguish distances between peers on the same circle. Depending on the circumstances on the network and on the surroundings, two peers on the same circle might be even one hundred meters apart from each other.

Radar view receives references to users with `user_appears`, `user_disappears` and `user_changes` methods. Using these methods it holds a list of users and their coordinates on the radar. Only pieces of information of the user used by the radar view are nick, profile picture and hop count. There is no underlying non-UI plugin for radar to hold more state information. Radar view is a pure UI plugin.

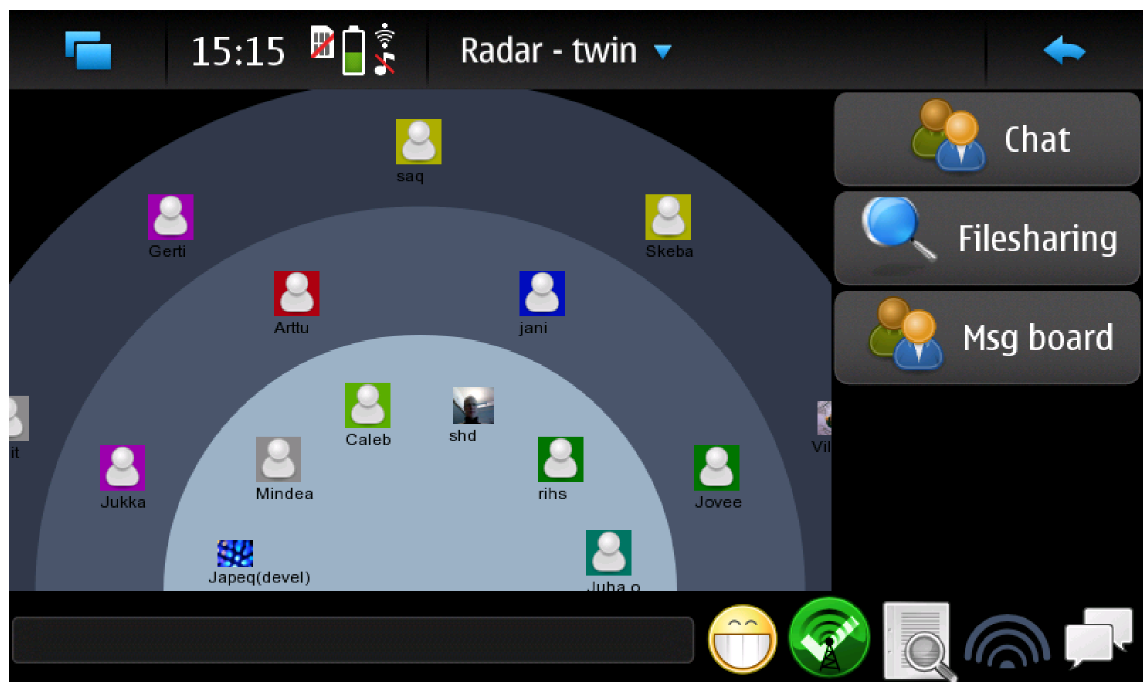

Figure 6.5. Radar view with many users.

The view was not implemented in a very sophisticated way, but only used GTK+ library's drawing functions. This led to the view taking up a lot of CPU time, which led to using large amounts of energy. With 10 peer on the view, 40% of CPU time was used for updating. This would be a problem if the view is used for long times, but if the view is not visible on the display, no drawing functions will be called and no CPU will be wasted.

6.4. Filesharing

Filesharing view allows user to publish media, browse and search content published by others, upload media directly to peers and stream media directly from other devices.

Media can be published to the network by selecting "Filesharing" and then "Publish". The content will be published to the selected community. If no community is selected, the content is targeted to everyone, that is, to the "twin" community. Adding published content is presented in Figure 6.6.

Shares can be browsed by selecting the "Browse" alternative from the filesharing dialog. Again a community or a user can be selected to filter the browsing results. Browsing content is shown in Figure 6.7.

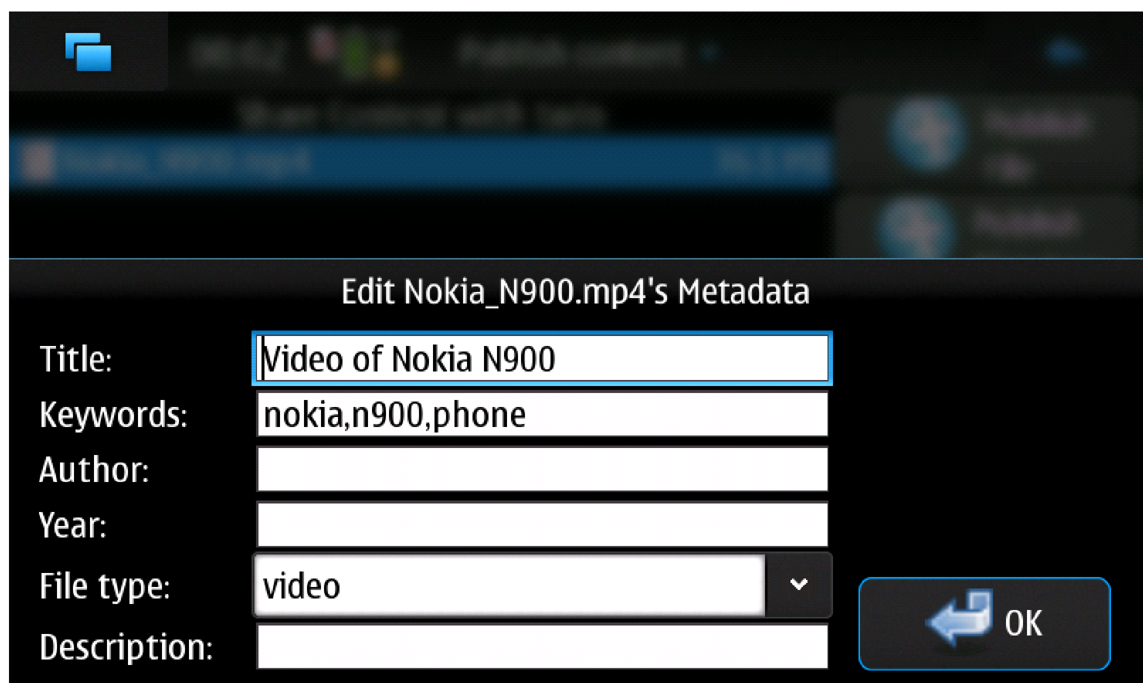

Figure 6.6. Filesharing view, for adding files, showing metadata of an added file being edited.

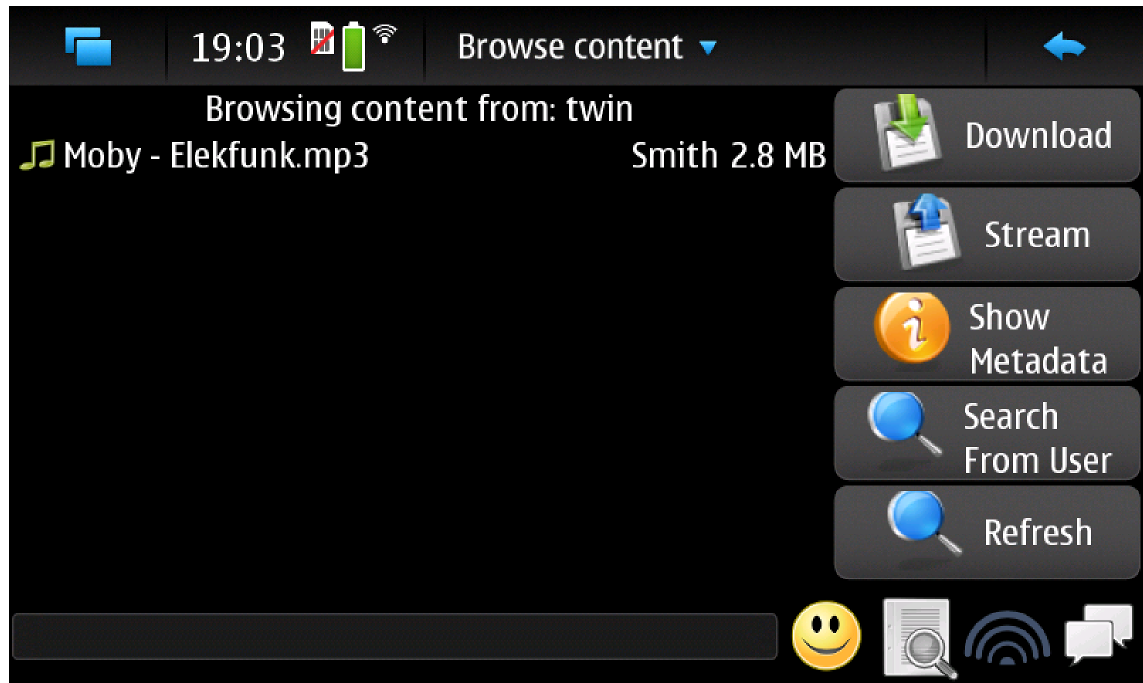

Figure 6.7. Filesharing view, showing one file ready for download.

Media can be downloaded, or streamed directly, if the file type is suitable for streaming. Streamable file types include music and videos supported by the device.

Published media may include metadata, such as title, author, file type, description and keywords. This metadata is used when published content is searched.

When downloading media, the devices must be directly linked to each other. Downloading is not possible with multi-hop networking. Browsing results are not filtered to only show published files from those, who are near enough. Instead, an icon to present *reachability* is shown. Reachability is here defined as meaning the likelihood of the file transfer succeeding. If the devices are more than one network hop away from each other, a warning icon is shown to inform the user, that the download may fail. Because there is the possibility of the network not routing optimally, the download may still succeed, so trying is not prevented.

Content can also be uploaded directly to other users, by selecting "Send file" from the filesharing dialog. Targets work also with this option: files can be sent to everyone in the network, to a certain community, or to a single user. Uploading requires one connection per targeted user, so uploading files to large communities is limited by the WLAN bandwidth.

Interfaces for the filesharing view are implemented in two parts: one module implements browsing and publishing shares, and another module implements an interface for following the status of the transfers. The views use interfaces from the filesharing

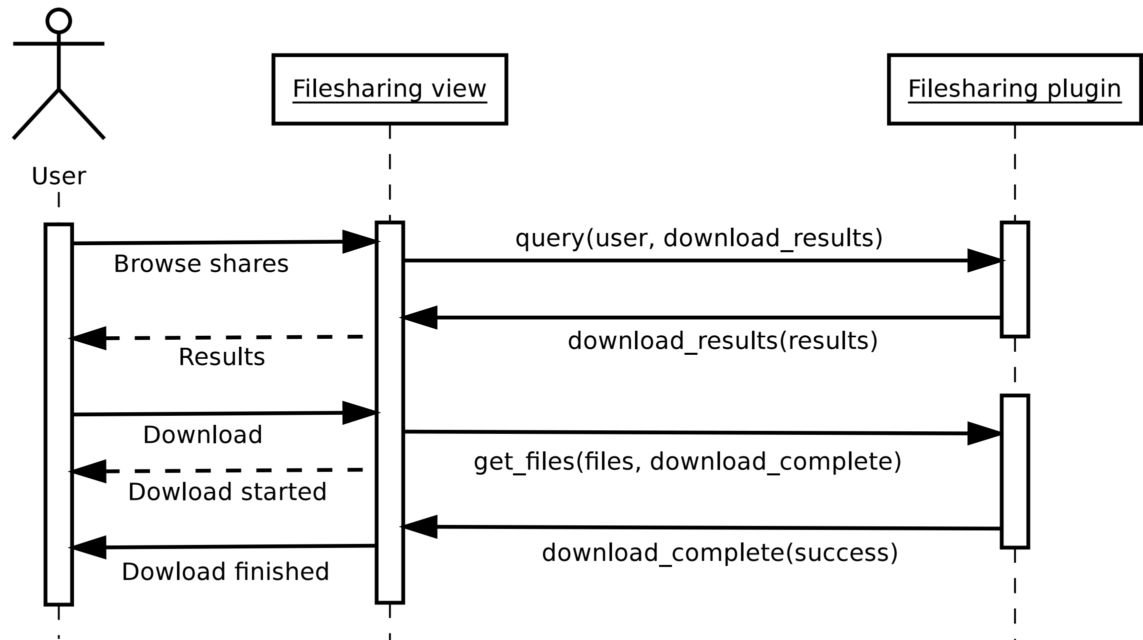

Figure 6.8. Filesharing download sequence. User first queries shares and gets a list of results. Then a share is selected to be downloaded. After the download is completed, user is informed.

plugin to publish, download and receive shares. The filesharing plugin holds all of the state information regarding the transfers, which is then used by these views. State of the user interface is changed using callback functions, which are called by the filesharing plugin.

Sequence diagram of downloading a share showing interaction between filesharing plugin and the user interface is presented in Figure 6.8. User first initiates to activity by clicking the ”Browse” button on the user interface. A list of results is asked from the filesharing plugin with `query` method. Nothing is returned, but the call includes a callback function `download_results`, which is used to pass the results after the query has been made. Then the user selects a file to download. Filesharing view user `get_files` method to download the share. Again a callback function, `download_complete`, is given. After the download is completed successfully, or the download failed, the callback function is called, and the user if informed of the status.

6.5. Chat

TWIN supports public group chats and private one-to-one chats. Group chats take place in communities, while private chats are between two users. Type of the chat is determined automatically by the selected target. A group chat in the ”twin” community is shown in Figure 6.9.

Chats are divided into tabs in the user interface. When a new chat message occurs, the tab will turn red to alert the user.

As chats usually occur in real time with short intervals between messages, chat feature was very vulnerable to the problems caused by unreliable transfer layer. Transmission delay of several seconds was immediately noted by the users, and missing messages caused a lot of irritation.

Some user interface features were designed to mitigate this. Unsent messages are marked, so that the user knows what messages from him the others have seen. Because of the functionality of the network layer, however, this is only a guess. For example, the message might have been delivered successfully, but the acknowledgment response was dropped.

Chat user interface module holds all of the state information for active chats. Messaging plugin, which is the underlying layer for the chat plugin, has no concept of a ”chat”. It only knows how to send and receive messages. New active chat is created when a message with a previously inactive target is received. Chats are closed only by the user. Messages are received through `new_message_cb` method, which is called by messaging plugin for every new message.

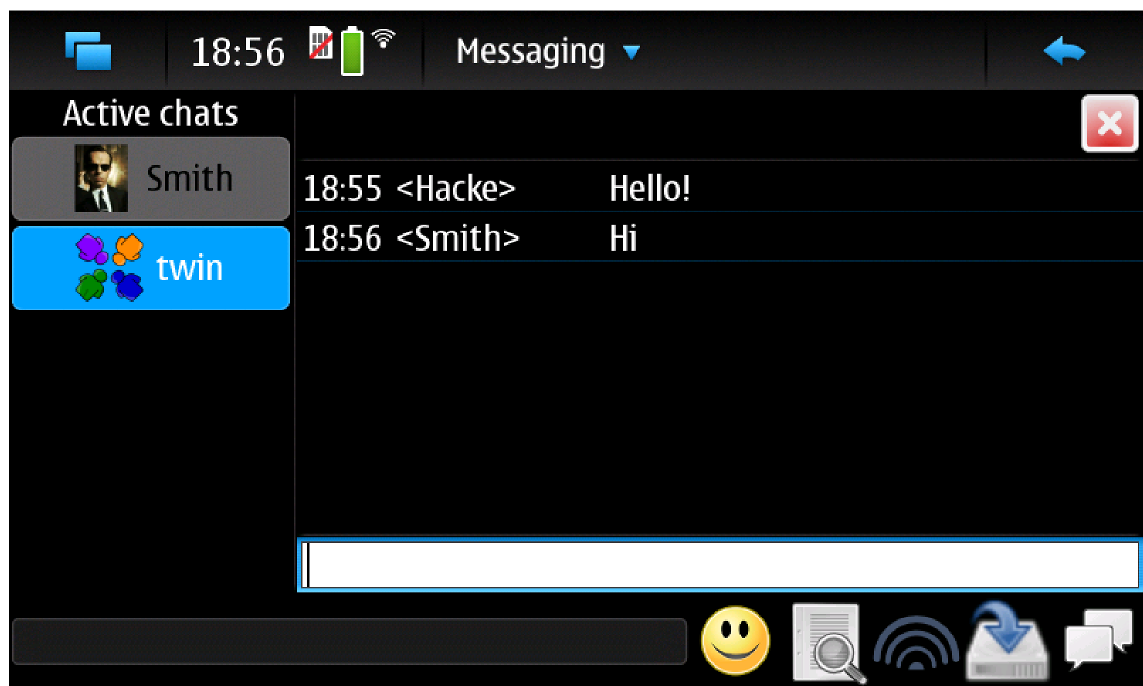

Figure 6.9. Chat in the public twin community, with a private chat open in the background.

6.6. Message board

The message board acts as personal bulletin board that every user carries with them. Purpose of it is to permanently hold text messages, which can then be viewed by other users at any time. This is as a contrast to chat messages, which can not retrieved again after they have been sent to the network. Example of the message board is shown in Figure 6.10.

Message board functions so, that the messages users write to it are stored to their own device. New messages are advertised to those who are currently active in the network. Older messages can be retrieved by querying messages from other devices. Every device replies to the query with the list of their messages. Every device also holds a cache of earlier seen messages. This speed up searching and browsing of the messages by reducing network round-trips.

Messages can be marked to belong to a certain community to limit the intended audience of the message. Messages are public, and every message is always published to all other users, but messages are filtered in the user interface based on which community the message is marked to belong.

The user interface plugin simply displays messages received from the network, and offers an interface to write new ones. Messages as retrieved from the messageboard

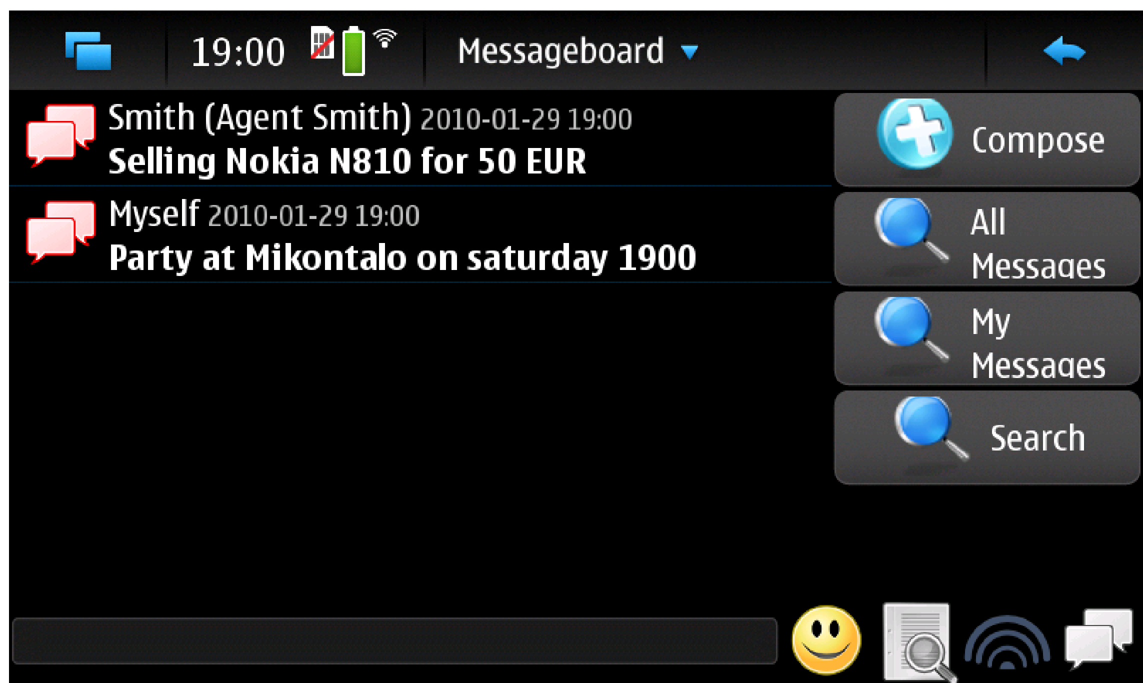

Figure 6.10. Message board with an advertisement of an N900 and an invitation to a party.

plugin with `query_messages` method. New messages are published to other users with `publish` method of the messageboard plugin.

6.7. Summary of the prototype interfaces

Almost half of the code written for TWIN was used to create user interfaces. While the views must hold some duplicate information with their underlying plugins, most of the displayed information is retrieved through different interfaces, reducing duplicate work and data.

This chapter presented the user interfaces that implement use cases developed for TWIN. In addition to these, there is much user interface code to implement additional parts of the application, such as dialogs, settings, helper code for graphics etc.

Table 6.1. gives a summary of the prototype interfaces in terms of lines of code (LOC).

Table 6.1. Main user interfaces implemented for TWIN.

View name	Module	# LOC	% LOC	Description
Community view	community_gui	2326	16.6	Visualize communities and users
Profile view	gui_user	515	3.7	Profile visualization and editor
Radar view	radar	243	1.7	Visualize network distance of users
Filesharing view	filesharing_gui	821	5.8	Browsing and downloading shares
	filetransfergui	255	1.8	Monitoring transfers
Chat	messaging_gui	354	2.5	One-on-one and group chats
Message board	messageboard_gui	407	2.9	Persistent messages
Total		4921	35.0	
Additional UI code		1490	10.6	
Total UI code		6411	45.7	
Total lines of code		14043	100	

7. PROTOTYPE TESTING

Even with a prototype program, checking of the correctness of the code and evaluating usability is crucial. While developing, only the code could be checked. Evaluation of usability in terms of does the program do what the user thinks it does, rather than does to program function as it was intended, requires a large enough group of test users.

7.1. Methods

We used a number of methods to help test the software. We created a PC version of the application to be able to move the evaluation task from the device to the PC where the development was done. We used assertions to ensure that the assumptions made were holding. We used linting software to find common sources for bugs, and lastly, a debugger software specialized for Python was used to track the source of the bugs to be able to fix them.

7.1.1. PC version

At first, TWIN was developed for and tested only on the N900 device. This was soon found to be very time consuming. Copying the software from the workstation to the device and running the program on the device took around 30 seconds for each try. Figure 7.1. illustrates this. As normal debugging workflow usually requires constant retries, this time penalty was too long.

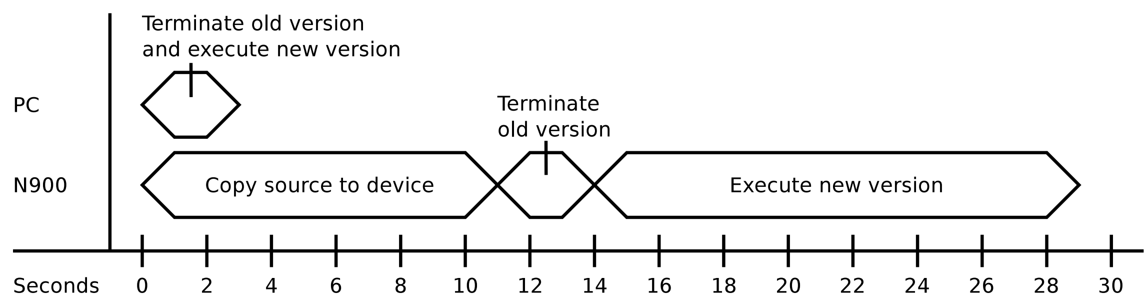

Figure 7.1. Timeline for initial steps of testing a new version on the device.

Programme 7.1. Example of detecting the run-time platform and creating a different button widget depending of the result.

```
1  import gtk
2
3  have_hildon = True
4  try:
5      import hildon
6  except ImportError:
7      have_hildon = False
8
9  def new_button(label):
10     if have_hildon:
11         button = hildon.Button(
12             gtk.HILDON_SIZE_AUTO_WIDTH | gtk.HILDON_SIZE_FINGER_HEIGHT,
13             hildon.BUTTON_ARRANGEMENT_HORIZONTAL)
14         button.set_title(label)
15     else:
16         button = gtk.Button(label)
17     return button
```

As both Python and GTK+ are available for both x86 and ARM architecture used on N900, it was decided that the prototype will be made functional on the PC. This effort required only minor changes to the source code, mainly taking into account the Hildon desktop environment on the N900. The source code used on both platforms is the same, but some run-time decisions on execution path must be made differently on each platform. Example session of the PC version running on 64 bit x86 hardware and a regular Linux desktop is shown in Figure 7.2.

Hildon uses custom widgets to create a touch screen enabled user interface. The widgets that had to be fitted for TWIN were file chooser, scroll area, text view, button and text entry. Programme 7.1. gives an example of how to create a different button widget depending on the platform.

PC version introduced also some problems to the development. While the target platform was N900, the functionality of the PC version was crucial to the development. Also, we did not want to maintain different versions of the code apart from the few wrapper utility functions. Because of the differences between pure GTK+ and Hildon, some compromises had to be made to the user interface design.

Other problem was that, when developing on the PC, evaluation of the user interface appearance on the device was sometimes forgotten. Changes that seemed minor on the PC sometimes had major effects on the device.

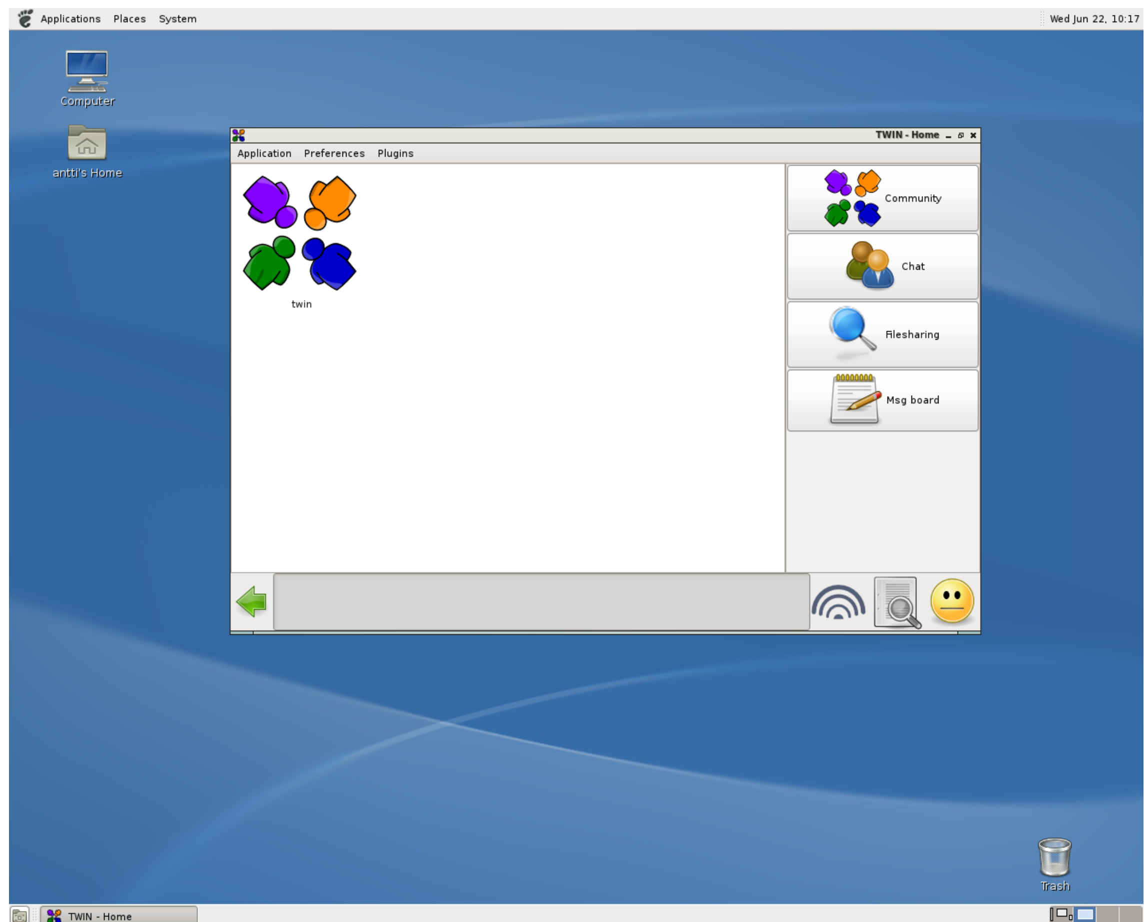

Figure 7.2. TWIN running on a PC under 64 bit Gentoo Linux and Openbox window manager.

7.1.2. Assertions

One tool used to automatically evaluate the correctness of the code were assertions. Assertions are logical predicates, which should always hold true when the expression is ran. If the assertion fails, meaning that the value of the expression is false, the program will be immediately terminated, and the offending line is reported.

Most bugs found with the use of assertions involved parameters with wrong types being passed to methods. In Python, there is no type checking, but a variable is considered suitable, if it possesses the member it was asked for. Also, in many cases in TWIN, functions can take multiple types by design. This reduces copying code and inventing new method names, which can introduce bugs to the code, and makes the code harder to maintain.

Another situation where assertions were found very usable was refactoring code. When interface changes are made, every piece of code calling the changed functions must be changed correspondingly. Assertions helped to locate unchanged code.

7.1.3. The Python Debugger

The Python Debugger (*pdb*) [22] is an interactive debugger for Python. It has a similar command line interface as the GNU Project Debugger [21]. Pdb includes features for settings breakpoints, stepping the program line by line, printing stack frames and running Python code using the variables under current scope.

Pdb is used to help trace bugs. Normal workflow for pdb is to make an intelligent guess about the origin of the bug, and set a trace to a suitable place before the bug is expected to occur. The code can then be stepped line by line to check to behavior. Another way of using pdb is to inspect the circumstances that led into the termination of the program. Pdb can print the stack trace and last values of the variables in the scope of the stack frame.

IPython [18] is a replacement shell for the regular Python shell, with more features, such as completing words by pressing tabulator key, syntax coloring, better support for writings inline python code blocks, and pdb. IPython's pdb integration can be used by simply importing `ipdb` instead of `pdb`. This way all of the IPython's features will be made available when the debugger is executed. Example session with IPython and pdb is shown in Figure 7.3.

```
antti@lelulaatikko:~/twin-antti
/home/antti/twin-antti/twin.py in <module>()
     11         import splash
     12         splash.splash_show()
     13
     14 import main
---> 15 main.main(options, args)

/home/antti/twin-antti/main.pyc in main(options, args)
     36         twinstate.load_external_plugins(options=options)
     37
---> 38         plugins_read()
     39
     40         listener.init()

NameError: global name 'plugins_read' is not defined
> /home/antti/twin-antti/main.py(38)main()
     37
---> 38         plugins_read()
     39

ipdb> plugin
plugin_cleanup    plugins.pyc       pluginstate.py
plugins.py        plugins_ready     pluginstate.pyc
ipdb> plugins_ready
```

Figure 7.3. Python debugger running on iPython shell, showing the last file and the last function call, a `NameError` exception caused by a missing letter, and the correct form found from the current namespace.

7.2. Problems

Problems complicating evaluation were mostly related to the fact that Python is an interpreted programming language. Because of that, most bugs can be found only on run-time.

Other major problem was, that there are no well established methods to automatically test graphical user interface of the software. As most of the user interface bugs relate to rendering wrong information, automatic unit tests and regression testing is hard.

7.2.1. Interpreted and dynamic

Interpreted language means, that the source code is taken as such and ran by an interpreter software, taking expressions and translating the into operations or system calls. This is as contrast to compiled languages, where the source code must first by compiled into machine code, which is then ran directly by the CPU.

The lack of compilation process, and thus the lack of compile time checks, introduces problems to testing. Only syntax checking is done before run-time, or as the abstract syntax tree (*AST*) is being constructed from the source code. Table 7.1. lists some

Table 7.1. Differences of automated error checking between an interpreted language with dynamical typing and a compiled language with static typing.

Check type	Interpreted	Compiled
Type checking	Run-time	Compile-time
Name checking	Run-time, or with a linting tool	Compile-time
Range checking	Run-time	Run-time
Syntax checking	Before run-time or with linting tool	Compile time

of the differences between dynamically typed interpreted languages and statically typed compiled languages.

Dynamic typing means, that the types are evaluated on run-time. This leads to the situation, where the correctness of the types must be checked by the programmer. In many places, the source code of TWIN is knowingly written to use multiple types of variables. These kind of situations can not be checked even with a linting tool able to infer types from the code path.

A linting tool is a program that does static analysis to the source code, i.e. without actually running it, to find common mistakes and bad quality code. Pylint [54] is a linting tool for Python. It checks the source code for style, bugs, and code structures prone to create bugs.

7.2.2. User interface testing

Testing graphical user interfaces proved hard. There are few tools for testing interfaces automatically, e.g. LDTP [33], Dogtail [30], and Sikuli [11]. These tools have many restrictions and require special techniques, such as accessibility enabled interfaces, to be able to run tests. Some of the tools, LDTP and Sikuli, work mainly by comparing screenshots taken of the application.

These techniques do not fit well for testing complex multi-user network software, where the interactions between user initiated events cause a great deal of randomness to the program flow. Events could be simulated to to remove this randomness to some extent, but that would not correspond to the real use cases.

Because of these difficulties, automated tools were not used to evaluate TWIN. Instead user interface code was evaluated by hand with repeated attempts to induce and repeat bugs. Still, many of the user interface bugs were only found during the user pilot with 250 people using the application.

8. EVALUATION

Results for this thesis consists of two parts. First the usability of the user interfaces is evaluated using using the experiences received from the TWIN pilot [56]. Then the effectiveness of the methods used in development is evaluated by comparing the amount of work taken to develop TWIN to the approximation given by the amount of source code lines.

8.1. Usability of user interfaces

In the pilot, 250 users used TWIN for two months. Users were asked to give feedback on the functionality and usability of the application. User interfaces seemed to be well accepted, based on the notice that there were little complaints about them. Two main complaints were about the unclear purpose of radar interface, and about messages being missed due to TWIN giving no notice at all when being left to the background, or when the phone was not actively used.

Latter was fixed during the pilot. The vibration plugin introduced in Chapter 5 was added. The plugin vibrates the phone when a new private chat message is received, or when a user marked as a friend appears in the network. Radar view was left as it was, as according to feedback, most users seemed to like it. Better user documentation could have helped with the issue. Radar view was mentioned as the most interesting single user interface feature in TWIN.

Users were asked to mention the most interesting feature in TWIN. Use cases mentioned most often were the ability to communicate with the people physically nearby, sharing content with then and seeing who is around you. The most often mentioned user interface feature was the radar. Most interesting features according to TWIN pilot are listed on Table 8.1. [56]

8.1.1. Ideas for improvement

During the pilot, the participants for able to give ideas for improvements through various ways. These ideas were gathered together, and the most often mentioned ones are presented in Table 8.2.

Table 8.1. Most interesting features of TWIN according to pilot participants.

Features	Mentioned by %
Communication with nearby people	21
Content sharing	19
Seeing who are around you	15
Radar view	12

Most ideas were related to the social aspect of the program. Three most often mentioned ideas were short games to play with strangers, more features that would help use TWIN for dating, and support for features that could be used by large crowds of people, such as a message wall, where visitors could post their messages in public events, or a collaborative painting canvas.

8.2. Methods

The methods used to develop TWIN were seen as functional. Prototypes could be completed fast without any formal specifications, and new ideas arising from them could be implemented through an iterative process. We were able to make many bugs fixes according to the given feedback by the pilot participants. Seven new versions were released during the pilot in addition to the original one.

It is hard to give numeric values for the methods. Constructive Cost Model [35] can be used to estimate the development effort that should have been needed to create TWIN. The model uses the amount of source code lines to estimate needed effort by using coefficients discovered by examining software projects of different sizes.

According to this method the needed development effort in order to create TWIN would be 3.3 person years. This is close to the realized development time.

Table 8.2. Ideas for new features and use cases gathered from the pilot.

Feature	Mentioned by #	Description
Games	37	Quick games to play with unknown people
Dating	29	More support for meeting new people, more content on profile pages
Happenings	26	More crowd features, e.g. a message wall
Advertisements	26	Location-based advertisements for companies
Localized info	10	E.g. warnings about traffic

9. CONCLUSIONS

This thesis presented use cases and user interfaces to implement those use cases in a social mobile ad hoc network application TWIN, as well as tools used to implement and evaluate the application, and problems that were faced during the development.

The main result of the project and this thesis was the application itself, including the user interfaces. Secondary results were the use cases, and experiences from tools and methods to rapidly prototype different ideas.

Objectives of the project were to create user-friendly user interfaces, which scale to hundreds of users, while taking into account the limited resources of the N900 mobile computer. The resource objective was met in the sense, that the bottleneck was the network layer, not the application. There were at most 150 people visible in the network to each other at once, with no observable problems to the application. User friendliness was studied in the pilot, and the users seemed to accept the user interfaces well.

Methods for testing and evaluation of the user interfaces were insufficient. While the PC version and its portability to the device helped to reduce the time spent for testing, there were no automated methods to test the user interfaces. Trying the application by hand is inadequate to find bugs. Interactions between the actions initiated by multiple users locally and over the network also complicated the evaluation process.

The project raised multiple ideas for future work. These have in part already been implemented in following projects. Some of the ideas include integration of TWIN into the phone, and usage of other social networks as a source for additional information of the users. New use case ideas from the pilot have not been implemented, and they should be studied in the future.

BIBLIOGRAPHY

[1] 3GPP. General packet radio service (gprs); service description. 2000.

[2] A. Ahtiainen, K. Kalliojarvi, M. Kasslin, K. Leppanen, A. Richter, P. Ruuska, and C. Wijting. Awareness networking in wireless environments. *Vehicular Technology Magazine, IEEE*, 4(3):48–54, September 2009.

[3] A. Beach, M. Gartrell, S. Akkala, J. Elston, J. Kelley, K. Nishimoto, B. Ray, S. Razgulin, K. Sundaresan, B. Surendar, M. Terada, and R. Han. WhozThat? evolving an ecosystem for context-aware mobile social networks. *Network, IEEE*, 22(4):50–55, July 2008.

[4] Kent Beck. *Extreme Programming Explained: Embrace Exchange*. Addison-Wesley Professional, October 1999.

[5] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the eleventh ACM Symposium on Operating systems principles*, SOSP '87, pages 123–138, New York, NY, USA, 1987. ACM.

[6] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4 (2nd Edition)*. Prentice Hall, February 2008.

[7] SIG Bluetooth. Specification of the bluetooth system, version 1.1. 2001.

[8] Steve Burbeck. *Applications Programming in Smalltalk-80: How to Use Model-View-Controller (MVC)*. Softsmarts, Inc., 1987.

[9] Vern Ceder. *The Quick Python Book*. Manning Publications, second edition, January 2010.

[10] Scott Chacon. *Pro Git*. Apress, August 2009.

[11] Tsung-Hsiang Chang, Tom Yeh, and Rob Miller. Project sikuli. http://sikuli.org/, 2012.

[12] Tom Christiansen, Brian D Foy, Larry Wall, and Jon Orwant. *Programming Perl*. O'Reilly Media, 4th edition, February 2012.

[13] Bram Cohen. The BitTorrent protocol specification. http://www.bittorrent.org/beps/bep_0003.html, June 2009.

[14] Nokia Corporation. Maemo. http://www.maemo.org, April 2011.

[15] Nokia Corporation. Maemo Software Development Kit. http://wiki.maemo.org/Documentation/Maemo_5_Developer_Guide/Development_Environment/Maemo_SDK, April 2011.

[16] Nokia Corporation. Nokia N9 touch screen smartphone – Specifications. http://europe.nokia.com/find-products/devices/nokia-n9/specifications, November 2011.

[17] Nokia Corporation. Nokia N900 Tech Specs. http://maemo.nokia.com/n900/specifications/, April 2011.

[18] IPython development team. Ipython. http://ipython.org/, 2012.

[19] Nathan Eagle and Alex Pentland. Social serendipity: Mobilizing social software. *IEEE Pervasive Computing*, 4:28–34, April 2005.

[20] Daniel Elstern et al. Maemomm. http://maemomm.garage.maemo.org/, April 2011.

[21] Free Software Foundation. Gdb: The gnu project debugger. http://www.gnu.org/software/gdb/, 2012.

[22] Python Software Foundation. pdb – the python debugger. http://docs.python.org/library/pdb.html, 2012.

[23] Python Software Foundation. Python programming language. http://docs.python.org/library/, September 2012.

[24] The Linux Foundation. MeeGo. http://www.maemo.org/about, April 2011.

[25] freedesktop.org. D-bus. http://www.freedesktop.org/wiki/Software/dbus, September 2012.

[26] freedesktop.org. Dbus bindings. http://www.freedesktop.org/wiki/Software/DBusBindings, September 2012.

[27] Magnus Frodigh, Per Johansson, and Peter Larsson. Wireless ad hoc networking – The art of networking without a network. *Ericsson Review*, 4(4):249, 2000.

[28] Doug Hellmann. *The Python Standard Library by Example*. Addison-Wesley Professional, June 2011.

[29] Edward Hervey. Gstreamer python bindings. http://gstreamer.freedesktop.org/modules/gst-python.html, November 2012.

[30] Vitezslav Humpa. Dogtail. https://fedorahosted.org/dogtail/, 2012.

[31] IEEE. Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification. (802.11), 2007.

[32] INdT. Pymaemo. `http://pymaemo.garage.maemo.org/`, November 2009.

[33] Eitan Isaacson and Nagappan Alagappan et al. Linux desktop testing project. `http://ldtp.freedesktop.org/wiki`, 2012.

[34] Bishal Raj Karki, Arto Hämäläinen, and Jari Porras. Social Networking on Mobile Environment. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*. ACM, December 2008.

[35] Chris F Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, May 1987.

[36] Steffen Kern, Peter Braun, and Wilhelm Rossak. MobiSoft: An agent-based middleware for social-mobile applications. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, volume 4277 of *Lecture Notes in Computer Science*, pages 984–993. Springer Berlin / Heidelberg, 2006.

[37] Janne Kulmala, Antti Laine, Marko Hännikäinen, and Heikki Orsila. Design for device-to-device communication for social networking. 2012. Submitted for publication.

[38] B.P. Lientz and E.B. Swanson. Software maintenance management: A study of the maintenance of computer applications software in 487 data processing organizations, 1980.

[39] Anderson Lizardo et al. PyMaemo. `http://pymaemo.garage.maemo.org/`, April 2011.

[40] Cameron Newham and Bill Rosenblatt. *Learning the bash Shell*. O'Reilly Media, 2th edition, January 1998.

[41] Tom Nicolai, Eiko Yoneki, Nils Behrens, and Holger Kenn. Exploring social context with the Wireless Rope. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, volume 4277 of *Lecture Notes in Computer Science*, chapter 112, pages 874–883. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[42] Nokia Corporation. *Fremantle Master Layout Guide*, October 2009.

[43] Donald A. Norman. *The design of everyday things*. Basic Books, 2002.

[44] Petru Paler, Ross Cohen, and Bram Cohen. Bittorrent-bencode. http://bittorrent.com/, July 2007.

[45] Stephen R. Palmer and John M. Felsing. *A Practical Guide to Feature-Driven Development*. Prentice Hall, February 2002.

[46] A-K Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, and C. Diot. Mobiclique: Middleware for mobile social networking. In *WOSN'09: Proceedings of ACM SIGCOMM Workshop on Online Social Networks*, August 2009.

[47] The GNOME Project and PyGTK Team. Pygtk. http://pygtk.org/, April 2011.

[48] Ryan Paul, Ars Technica. Nokia's new MeeGo-based N9 is set up for failure. http://arstechnica.com/gadgets/news/2011/06/nokias-new-meego-based-n9-is-set-up-for-failure.ars, June 2011.

[49] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press, March 2004.

[50] E. Seidel. Technology of high speed packet access (hspa). *NOMOR Research White Paper*, 2006.

[51] Software in the Public Interest, Inc. Debian GNU/Linux. http://www.debian.org/, April 2011.

[52] SourceForge.net. Proximate. http://sourceforge.net/projects/proximate/, November 2011.

[53] GStreamer Team. Gstreamer. http://gstreamer.freedesktop.org/, October 2012.

[54] Sylvain Thenault. pylint. http://www.logilab.org/project/pylint, 2012.

[55] Ian Vo, T. J. Purtell, Ben Dodson, Aemon Cannon, and Monica S. Lam. Musubi: A mobile privacy-honoring social network. http://mobisocial.stanford.edu/papers/musubi.pdf, September 2011.

[56] Kaisa Väänänen-Vainio-Mattila, Petri Saarinen, Minna Wäljas, Marko Hännikäinen, Heikki Orsila, and Niko Kiukkonen. User Experience of Social Ad Hoc Networking: Findings from a Large-Scale Field Trial of TWIN. In *MUM'10, 9th International Conference on Mobile and Ubiquitous Multimedia*. ACM, December 2010.

[57] Matthias Warkus. *Official GNOME 2 Developer's Guide*. No Starch Press, Inc., April 2004.