TAMPERE UNIVERSITY OF TECHNOLOGY

SANTTU LAKKALA

PROGRAM-LEVEL INTERFACE FOR THE WORMHOLE FUNCTIONALITY

Master of Science Thesis

Examiner: Tommi Mikkonen
Examiner and topic approved in
the Faculty of Computing and Electrical
Engineering Council
meeting on 9 December 2009

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
SANTTU LAKKALA: Ohjelmallinen rajapinta madonreikä-ominaisuuden käyttöön
Diplomityö, 45 sivua
Kesäkuu 2010
Pääaine: Ohjelmistotuotanto
Tarkastajat: Tommi Mikkonen
Avainsanat: vertaisverkot, mobiiliohjelmointi, madonreikä

Kilpailu modernien mobiililaitteiden markkinoilla on kovaa ja käyttäjät odottavat jatkuvasti enemmän laitteiltaan. Eräs merkittävimmistä tekijöistä on laitteen alustalle saatavien sovellusten määrä ja laatu. Viime aikojen huolet sosiaalisten verkostoitumispalveluiden yksityisyydensuojasta, yhdessä nousevien mobiili-Internet -nopeuksien ja laskevien hintojen kanssa, luovat mobiililaitteista houkuttelevan kilpailijan keskitetyille sosiaalisille verkostoitumispalveluille.

Vertaisverkkotiedonsiirto-ominaisuudet ovat hankalia toteuttaa, erityisesti mobiililaitteilla. Kehittäjiä voidaan houkutella tekemällä tiedonsiirto-ominaisuuksista helppoja toteuttaa ja käyttää. Kehittäjät voivat myös toteuttaa sovelluksia, joita ei muille alustoille ole olemassa.

Tässä diplomityössä suunniteltiin madonreikäjärjestelmä helpottamaan sisällön jakamista tukevia sovelluksia. Suunnittelun paino oli ohjelmallisen rajapinnan helppokäyttöisyydessä. Suunniteltu järjestelmä piilottaa vertaisyhteyksien luonnin monimutkaisuuden yhden yksinkertaisen rajapinnan, joka tukee useita sisältömuotoja, taakse.

Suunniteltu rajapinta ja sen toteuttavat komponentit tekevät vertaistiedonsiirroista olennaisesti helpompia toteuttaa. Työssä rakennetun toteutuksen avulla tietoa voidaan siirtää useiden protokollien kautta, joista automaattisesti valitaan tehtävään sopivin.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Computer Science
SANTTU LAKKALA: Program-level interface for the wormhole functionality
Master of Science Thesis, 45 pages
June 2010
Major: Software Engineering
Examiner: Tommi Mikkonen
Keywords: peer to peer networking, mobile devices programming, wormhole

Modern mobile device market is highly competetive, and people are expecting more and more from their devices. One of the crucial factors is the number and quality of applications for the platform. Recent privacy concerns with social networking sites, coupled with the rising mobile Internet speeds at reasonable prices, are making mobile devices an attractive competitor to centered social networking sites.

Peer to peer data transfer connections are complex to use in general, and even more so on mobile devices. By making the data transfers easier and easier to adopt, developers can be attracted to the platform. Developers are also able to easily create applications not found on competing platforms.

In this thesis, a wormhole system was designed to ease the development of content sharing applications. The focus of the design was on the ease of use of the program level interface. The developed system hides the complexity of peer to peer connections under one simple interface, which supports a number of different content types.

The designed interface and implemented infrastructure successfully make data transfers easy to accomplish. The implementation supports initiating and transfering the data over multiple protocols, automatically choosing one that is best suited for the task.

PREFACE

I would like to thank Kari Oinonen and Mikko Terho from Nokia Devices for the possibility to make this thesis and for their valuable input in the design process.

I would also like to thank the examiner of the thesis, Tommi Mikkonen, for his suggestions, corrections and especially for the pushing me through the writing on time.

Last, but not least, I would like to thank my wife for her support, understanding and encouragement, especially during the last year of my studies; and my family for their support throughout my studies.

Tampere June 3, 2010

Santtu Lakkala
santtu.lakkala@nomovok.com

CONTENTS

1. INTRODUCTION

With the spreading of the Internet, collaboration and sharing have evolved into one of the main features of computer systems. Unfortunately most applications do not implement collaborative functions and the user is required to use other applications to establish this, like instant messengers or email. In the mobile realm switching between applications is often hard, making collaboration a lot harder.

Applications rarely implement these functions, because creating them is quite complex. It requires usage of contact book and protocol implementations, checking contact availability and lots of other things. It may also cause trouble with other applications that are trying to use the same accounts for other purposes.

In this thesis, we have designed and implemented a data transfer system for a mobile platform. The main goal of the design was the ease of use of the program level interface.

The goal of the thesis is to create an easy to use interface for programmers that need peer-to-peer data transfers. This will allow application developers to quickly and effortlessly create programs that communicate with the contacts of the end user. The communication can occur in real peer-to-peer manner, provided that a suitable transfer channel, like Zeroconf. They may also use centralized services to aid in creating a peer-to-peer connection, or all of the data may be transferred via a server. The important part is, that the application programmer does not need to care about it.

The problem with the existing interfaces is that they are complex to use and hard to understand. The complexity is reduced by making the wormhole system do as much of the work normally involved, leaving only the bare minimum to the application developer. The system will do service routing, i.e. choosing the best carrier; connection validation; and contact selection.

As a technical contribution, we have designed an API for the data transfers. Also a library implementing the API was implemented, together with a backend service that the library uses. In Chapter 2 we explain the concept in detail. In Chapter 3 we introduce the components that the implemented system interfaces with. In Chapter 4 we explain the reasons and methods for the API design. In Chapter 5 we present the results of the API design and explain the architecture of the implemented system. In Chapter 6 we evaluate the results.

2. CONCEPT OF WORMHOLE

In popular culture, a wormhole is a portal between two, usually distant places. Anything going in at one end appears at the other end nearly instantaneously. Anything going into the wormhole does not need to understand how it works, or where it goes.

A wormhole, in this context, is a tunnel for transferring data. The user of wormholes does not need to know the method of transport. Nor do they need to know the destination. Thus the analogy to the wormholes from popular culture.

The analogy is also used as the representation in the user interface (UI). Whenever wormholes are used to transfer data to a contact shown in the UI, a wormhole symbol appears on top of the contact. Additionally some kind of zoom effect to further illustrate the effect could be used.

2.1 Overview

The general idea of wormholes is shown in Figure 2.1. In the figure, there are three mobile devices, and the middle one has wormhole connections to the devices at the sides. The wormhole connections are denoted with the dark tubes; this is how the application sees the connection. The actual data transfer channel used underneath the wormhole connections are represented by the yellow lightnings. The connection between the device on the left hand side and the device in the middle is established directly using peer to peer connection. Sometimes it is not possible to establish a direct connection, and the data needs to be proxied through a relay. A relay server is represented by the black tower case at the top. Now the data transfer between the device in the middle and the device on the right hand side goes via the server, as shown by the yellow lightnings. The wormhole connection, however, does not show this, and it is virtually indistinguishable from the direct connection.

While in the figure, the endpoints of a wormhole are shown as devices, the endpoints are actually people, or contacts, not devices. A contact may be reachable through multiple protocols or accounts. A contact may even have multiple devices reachable at the same time. This exposes a limitation in the system, as a certain device cannot be targeted, even if wanted. However this is relatively rare requirement and a relatively small deficiency.

While peer to peer connections are always the preferred mode of transport, some

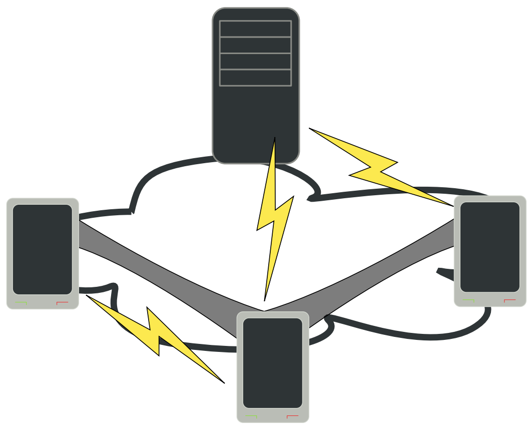


Figure 2.1: Wormhole enabled mobile devices communicating over the Internet

protocols or networks may prevent direct connections. That is why it is important to be able to do the transfer also in indirect ways. The wormhole system takes care of the path finding problems. It will use the transport channel most appropriate for the data at hand. The decision is based on details of the destination contact and the characteristics of certain protocols. The system uses details like presence information and estimated transfer speed.

In Figure 2.2 a typical home screen scenario is shown, where a note is being sent to a contact via a wormhole. Once the user releases the drag, the note is sent and will appear on the device of the contact. This kind of use case clearly shows the graphical representation and how it is used.

In this kind of use case, the application developer needs to know where the wormhole points to, or more precisely need to be able to indicate the target upon the creation of the wormhole. While this does break the name analogy and idealism behind the idea, it does give versatility to the system.

The contact viewer and wormhole UI application is not part of this thesis. The application is quite simple, and shows few most frequently used or user selectable contacts. When the user drags anything from the system onto the contact, the wormhole indicator appears, and on drop, the data transfer is initiated.

Another example is shown in Figure 2.3. A social media main screen widget that shares status updates over wormholes to pre-defined set of contacts. Implementation could be done by passing simple XML files over the wormholes. The XML files would be sent to a set of contacts defined in the widget configuration. Upon reception, the XML files would be parsed, and contents drawn to the widget.

A third example of a collaborative text editor is shown in 2.4. It allows end users to share a piece of text over wormholes. The users can then edit the file and all changes will be immediately visible at the other end. The users can also make notes

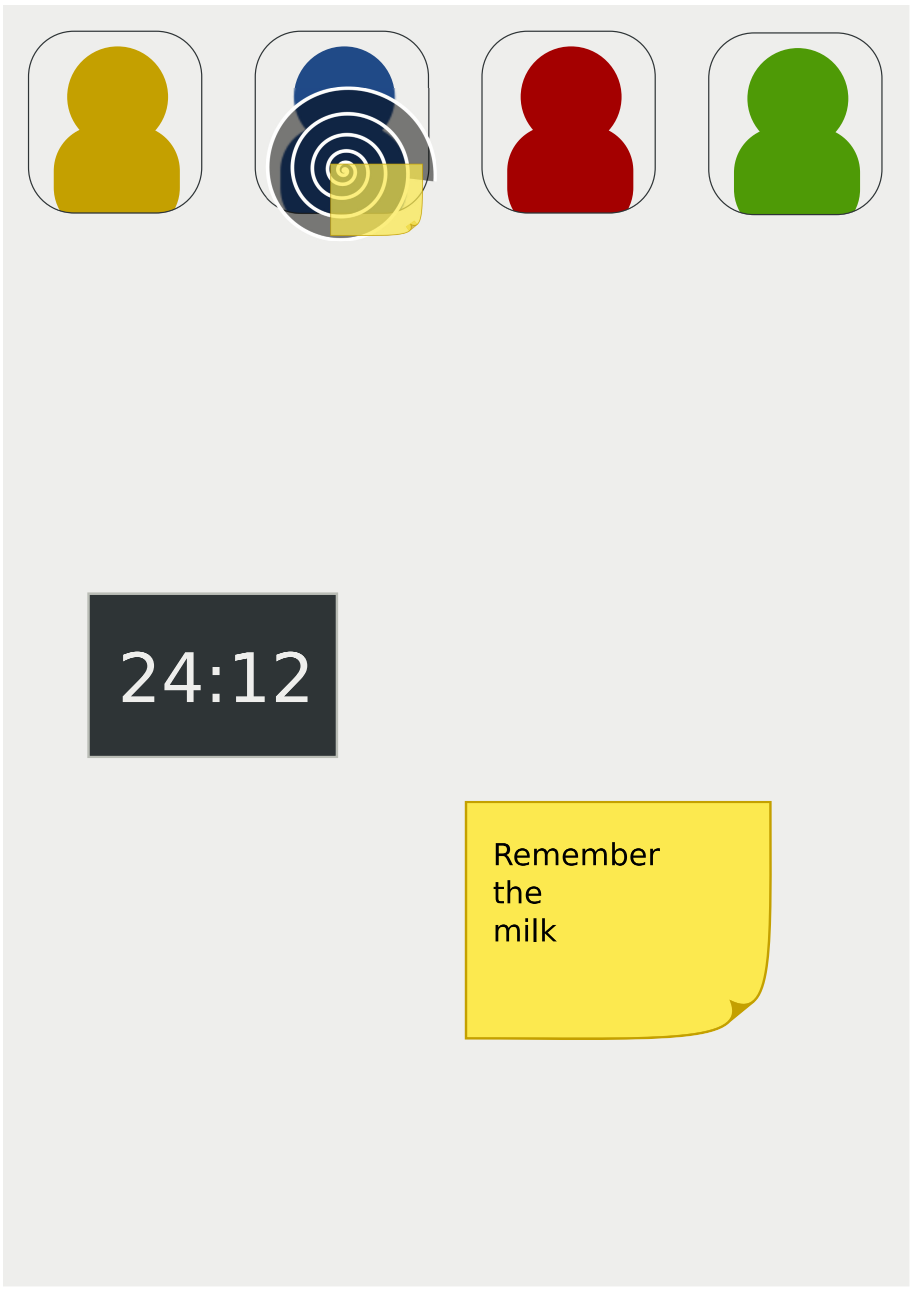


Figure 2.2: Wormhole usage on home screen on a touch screen device.

on any line in the file. In the image the text editor is used to find a problem in a source code file.

In general, wormholes should work in a peer-to-peer manner, transferring any data straight through the Internet. However, in certain situations it may be necessary to do the transfer through a server that proxies the data. This should not be apparent to the user of the API.

All data transfers between nodes should be encrypted, but in the scope of this thesis, the encryption is left to the underlying protocol(s). Most protocols do not

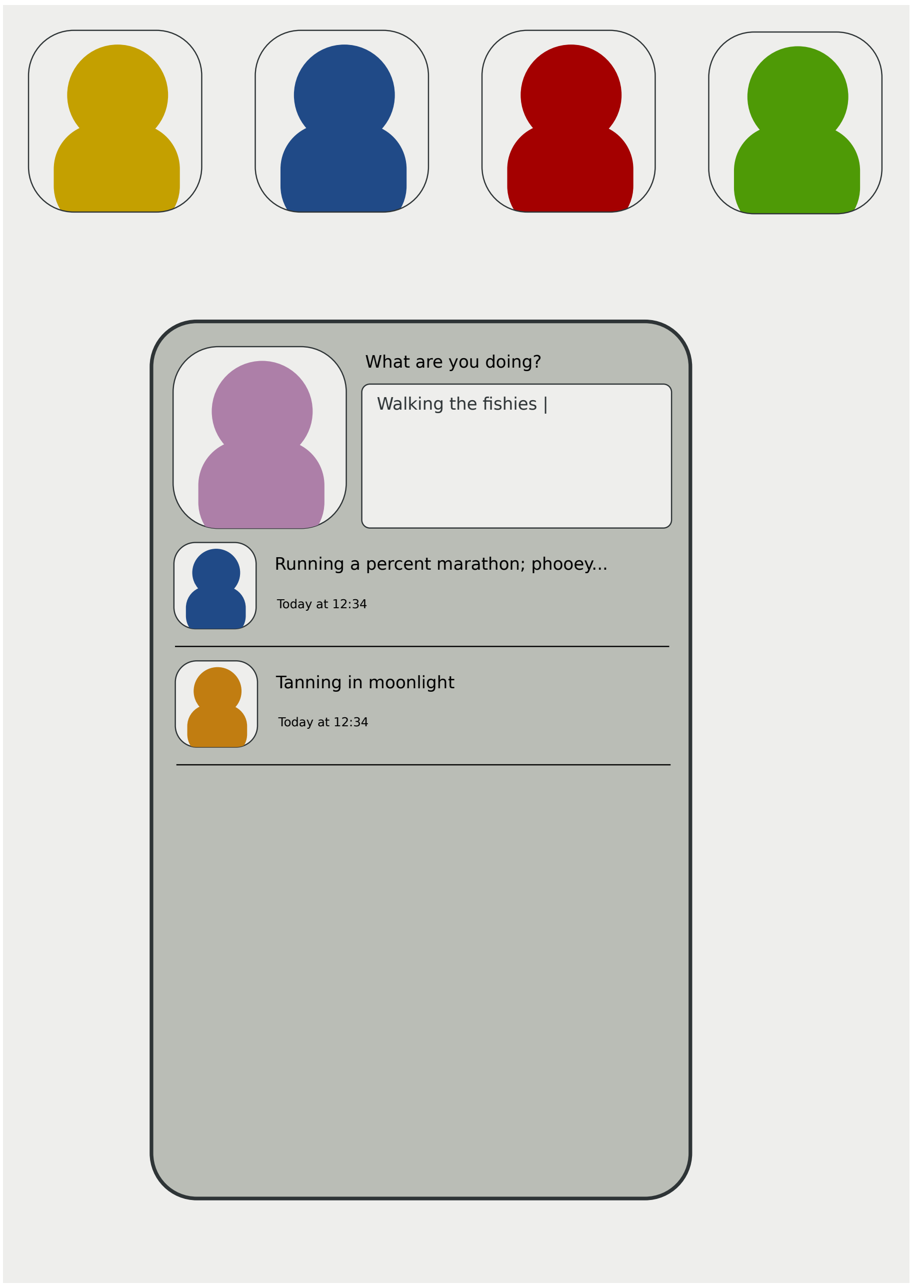
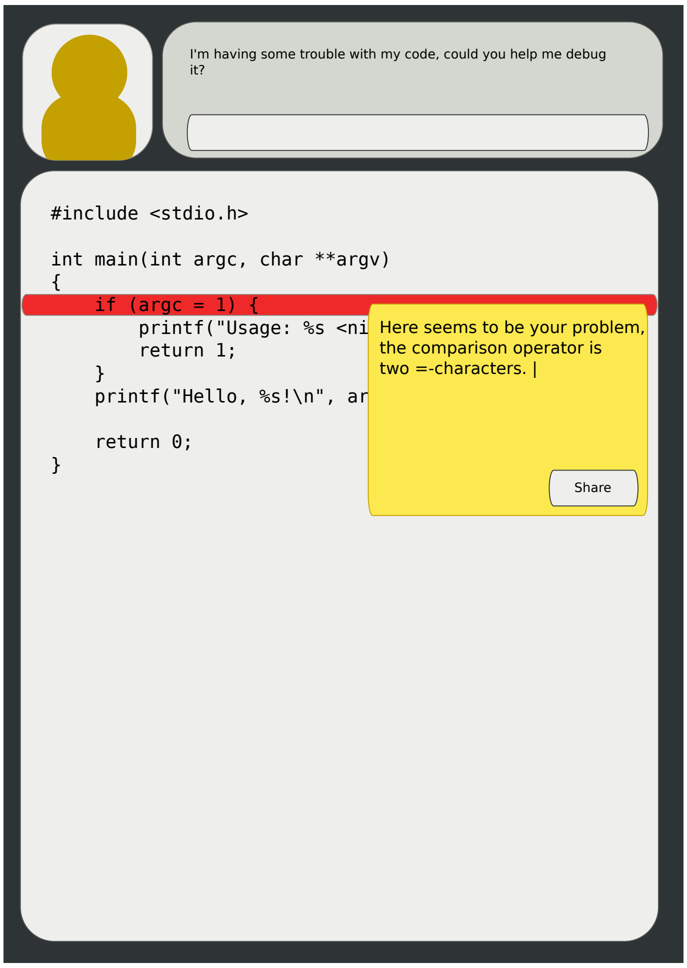


Figure 2.3: Wormhole based status sharing widget

encrypt the transfers.

2.2 Device to device wormholes

There are two main types of wormholes. In device to device wormholes, on which type this thesis concentrates on, the wormhole is opened to a contact. Any file or stream may be sent through the wormhole, no additional information is needed. When a file or stream is sent, the wormhole system decides the best action for the data based on its type. The receiving end then uses the meta data provided by the


I'm having some trouble with my code, could you help me debug it?

```
#include <stdio.h>

int main(int argc, char **argv)
{
    if (argc = 1) {
        printf("Usage: %s <ni
        return 1;
    }
    printf("Hello, %s!\n", ar

    return 0;
}
```

Here seems to be your problem, the comparison operator is two =-characters. |

Figure 2.4: Wormhole based collaborative text editor

sender to determine the action for the received data.

The device to device wormholes can be considered as representations of the contact, not actually the data transfer channel. The channel is only established once the data is known. This means that any data transfer through a device to device wormhole may fail if no means of transport can be established.

The note sharing, as shown in Figure 2.2, can be done using device to device wormholes. The note is serialized using JavaScript for example, and the serialized data is sent as a file. The whole widget, including its code, may be shared. The platform also contains an automatic installer and code injector, so that a JavaScript file may only contain the basic program logic, and the rest is automatically installed in the receiving end.

2.3 Program to program wormholes

The device to device wormholes are somewhat limited for certain uses. The streams are limited to unidirectional transfer, and the streams should have constant flow of data. Program to program wormholes allow almost any kind of transport. They are quite similar to normal sockets.

The main difference between program to program and device to device wormholes is that upon creation, the creator of a program to program wormhole needs to specify some program identifier in addition to the contact. The identifier can identify either the program, or an instance of it. Upon creation, the data channel is established right away. A program can presume that sending data through a program to program wormhole is unlikely to fail.

Program to program wormholes enable bidirectional data transfer. Either end of the wormhole may transmit data through the wormhole. In addition the wormhole may be used for signaling. Both ends may keep the channel silent and only send data when there is something to send.

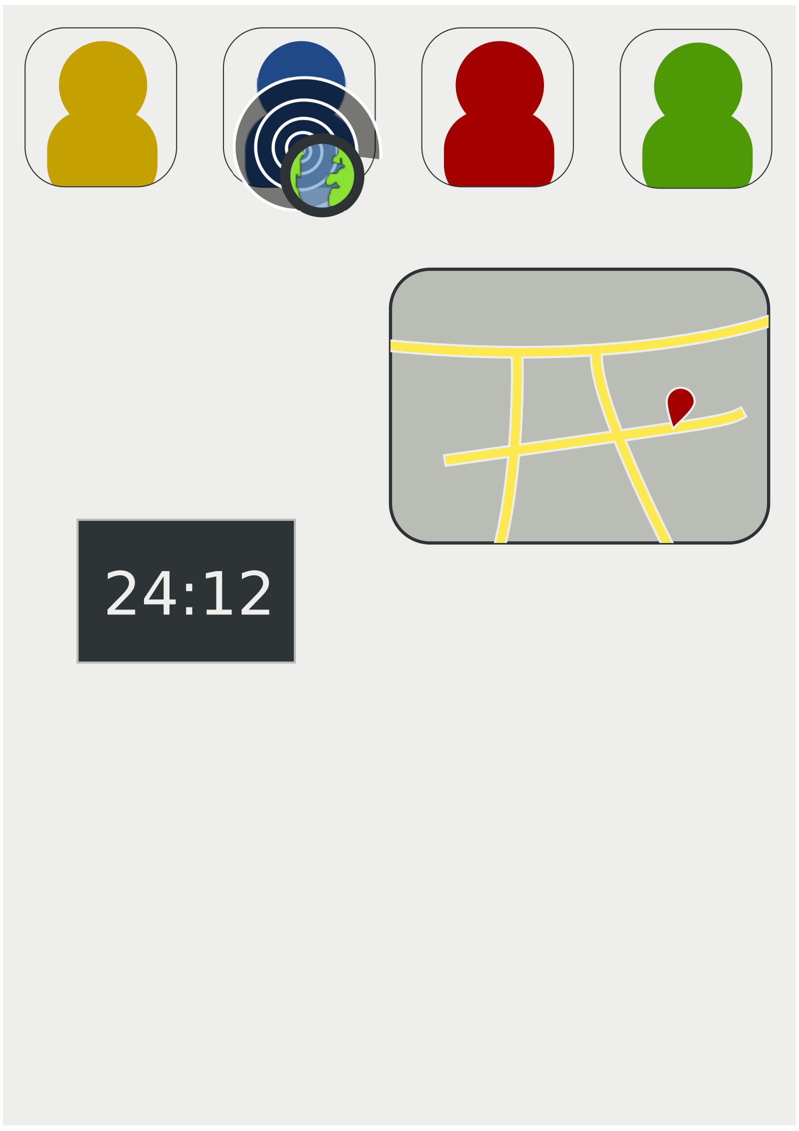


Figure 2.5: Wormhole usage on home screen on a touch screen device.

In Figure 2.5 a map widget that is being shared is shown. The sharing of the widget, including the current position can, and should, be implemented using device to device wormholes. If the location is wanted to be kept up to date over a longer period of time, the device to device wormholes are no longer a good fit. The information consists of small packages sent at unpredictable intervals. Even though it could be implemented as a stream, this kind of data transfer is better handled with program to program wormholes.

2.4 Wormhole API

The wormhole API is the interface for an application programmer through which he can send and receive files and streams. The goal is to make it possible for the programmer to not need to know what the system does, or how it works. They can just drop a file in, or broadcast a stream, and it will magically appear at the other end; much like wormholes in sci-fi movies.

From the perspective of the application developer using the API, the wormhole looks as illustrated in Figure 2.6. In the figure the Wormhole API is represtented by the Wormhole image. The Wormhole is a pipe, into which files and streams can be thrown, and they appear from a wormhole in some other place. There is nothing else for the developer to worry.

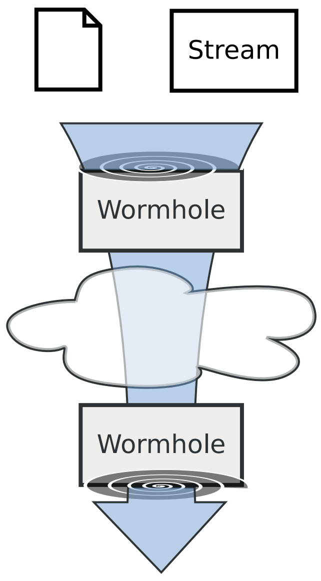


Figure 2.6: The concept of wormhole: files and streams are transferred between two endpoints.

3. STARTING POINTS

Our target platform is an operating system for mobile devices ranging from phones to small laptop computers. The operating system is based on Linux kernel and GNU userland tools, with Qt as the target framework for GUI applications.

As mobile platforms are usually limited in resources, they imply some requirements to the implementation. First, and most importantly, the system should allow the operating system to do power management by keeping activity at minimum when nothing is happening. In implementation terms this means that the system should never poll for changes, but rather use methods provided by the operating system to wait for events in idle state. Polling has dramatic impact on battery life, even at relatively low intervals.

The second scarce resource is memory. The importance of memory handling is even higher for long running processes. Memory management must be done well to avoid memory leaks. All memory should be freed immediately when no longer needed. Also processes should be kept down to minimum, they take up some memory just by themselves.

The third, by importance, is processor time. In terms of processing power, processors used in mobile devices are quite underpowered when compared to modern desktop computer processors. While saving processor cycles is not quite important enough to justify optimizations, certain precautions should be taken in design phase to ensure smooth operation.

3.1 Programming framework

The API is designed for Qt programs, which makes it sensible to write the implementation in Qt also. The implementation is written in C++, so bindings can be created for other languages that have bindings for the Qt framework.

3.1.1 Qt

Qt is a cross-platform application and UI framework written in C++ [11]. Qt has many useful features built into it that are beneficial for the implementation of the wormhole system.

The most important one is the Qt mainloop. The mainloop makes it possible to build asynchronous systems without the added complexity of threads, without creating own implementation of such multiplexing system. The mainloop is a system that sleeps until there is an event that requires attention, and then calls appropriate handlers for the event [12]. Using the mainloops for all processing is also good for power consumption, as the sleeping state is handled by operating system, which may use power saving methods when there are no processes actively using the processor.

The wormhole system also uses Qt's signal and slot system. Signals and slots provide an easy way to implement the observer design pattern, eliminating the need of listener interfaces. Signals are notifications that objects can emit, and other objects may register their slots to be run when a signal is emitted. [14]

Qt also includes support for D-Bus, a message bus system. It is integrated with the event system and receiving messages is possible whenever the control is in the Qt mainloop. Qt also provides adaptor classes created from XML-based instrospection data, which makes exporting objects for remote calling easy. Also Qt signals can be automatically emitted over D-Bus also.

Also included is a shared pointer implementation, QSharedPointer. Shared pointers are in concept similar to the auto_ptr of the C++ standard template library, but implement sharing the data with reference counting. Shared pointers automatically destroy the contained data when all pointer instances, or references, are gone [13]. Similar data structure is also planned for C++0x. [20]

3.1.2 D-Bus

D-Bus is a message bus system, a way for doing inter-process communication [5]. D-Bus is widely used in modern Linux distributions for communication of the core programs, as well as the applications the user uses. Unlike many other similar systems, D-Bus is binary protocol, and all data is sent using a binary representation, not readable text or XML.

Programs connecting to D-Bus have a service name. Each client is automatically assigned a *unique name* of the form ":1.23", which is unique in the lifetime of the bus. Programs may additionally request well-known service names. Each name may only have one owner at any given time, but one name may be owned by different clients at different times. The names are separated using dots, with similar naming convention as Java packages.

Exported objects in D-Bus are identified by an object path. The object path looks like a file path. Each object can have one or more interfaces, and each interface can have methods and signals. The methods and signals have a signature, which tells the argument and return value types of the methods and signals. A D-Bus method call may have multiple return values. An object path needs only be unique in the service, not in the bus.

The methods are remote calls to certain object. When doing a method call, the caller needs to know the service where the wanted object is running, and the path where it is located. In addition it needs to know the interface, method name and the argument types for the method. If any one of them is not known, the message will not be delivered. Signals, on the other hand, are messages that are sent to any client that has registered itself to listen for the signal. A signal has an object path and interface. The object path should be the same object path as where the sender of the signal is listening for method calls.

It is customary that the root object of a service is at object path "/". Using an interface name specified in the D-Bus specification, any object can be introspected, and they provide details of all implemented interfaces.

D-Bus only supports a set of primitive types and three types of collections. The primitive types include unsigned and signed integers in different sizes, floating point number, strings, object path and boolean. The first collection is an array, which can contain arbitary number of items. The second collection is a dictionary entry, which has a key and a value. An array of dictionary entries forms a map. The third collection is a struct, which can have a fixed number of members. There is also a variant type, which may include anything mentioned above. There are no NULL values in D-Bus; a variant must have a type and a value, and strings must have a value. Empty strings are allowed.

D-Bus has two busses. First one, the system bus, is usually started quite early in the operating system boot process. Hardware managers and the init process use it to communicate with each other. Access to the system bus is usually quite limited, allowing only certain users access certain interfaces and object paths. The services in system bus are run as privileged user rights, usually the administrator.

The second bus, session bus, is started whenever the user starts their session, usually when they log in. The address of the session bus is only known by programs in the session, and all services in session bus are running with the same privileges. The access to the session bus is usually not restricted, and one can do anything in the bus, given they have the address.

Both busses include an auto start mechanism. Whenever a message is sent to a service that is not running, and a meta file exists for the service, the bus daemon automatically starts the program. Then when the program starts up and registers to the bus, the message is delivered. This can be disabled when constructing the message, if automatic start of the recipient is not desired.

3.2 The communication framework

The platform has many components that the system must cooperate with. They are discussed in the following subsections.

3.2.1 Telepathy

Telepathy is a collection of well-defined D-Bus interfaces for protocol independent communication. The project also provides helper libraries to use and implement these interfaces for different programming languages and frameworks, and some protocol implementations using the aforementioned libraries.

The design makes the system quite robust. Different parts run in different processes and one process crashing does not take the whole system with it. It also gives high independence of specific programming language, parts can be implemented in any language that has D-Bus bindings. [9, section 1.1]

Originally the project aimed at supporting instant messaging and voice and video calls, but later evolved to include generic data streams, dubbed Tubes, and file transfers. However, due to the goals of the project, the usage of the framework is rather complicated thus the need for a simplifying layer.

Major users of telepathy framework are the Nokia Internet Tablet products [10], GNOME's Empathy [7] and OLPC's Sugar [15]. Telepathy is usable in the Qt framework via the Telepathy-Qt4 bindings provided by the Telepathy project.

Telepathy's APIs are very versatile, but also very complex. A file transfer example using telepathy-qt4 bindings [1] is about 500 lines of code. Most of the code goes to many different slots for asynchronous events.

By using standard Telepathy tubes, we gain compatibility with all other systems using Telepathy, but it is up to the receiving end to decide the action. It likely means that the receiving end will simply save the file after asking the user. As Telepathy uses standard file transfer metchanisms, the file transfer system is also compatible with other clients that implement file transfers. This, unfortunately, does not apply to streams, which are specific to the Wormhole system.

3.2.2 Mission Control

The centerpoint in the Telepathy system is the telepathy mission control daemon. It stores the accounts of the user, and handles bringing up the connections when requested. Usually this request comes from an instant messenger client, like empathy in GNOME. On the target platform, the best solution is to have a dedicated status setter, so the instant messenger client needs not be running.

The platform also needs a program that listens for contact status and avatar changes, and updates them to the address book. This is out of the scope of this work, and is assumed to already exist.

Unfortunately the TelepathyQt4-bindings do not yet provide any support for the mission control. This means that some kind of bindings need to be created or generated.

3.2.3 Supported protocols

The Telepathy framework is a multi-protocol system. It is extensible by writing connection managers for new protocols. At the time of writing there are connection managers for

- XMPP,
- Windows Live Messenger,
- Internet Relay Chat,
- Link-local XMPP,
- Skype and
- SIP.

There is also a connection manager that brings all protocols supported by libpurple [17], the protocol support library of Pidgin [16], a multi protocol instant messaging client. [6]

All of the protocols above can support file transfers: XMPP via the XEP-0096, IRC via DCC file transfers, SIP with MSRP and MSN has built in file transfer support. Only XMPP and SIP support data streams in a standard way. IRC DCC connections can be used, but it would be a proprietary extension.

The connection manager implementations vary in maturity. The XMPP manager, dubbed "Gabble" is the one getting most updates. The updates are moved to the Link-local version at somewhat slower pace. These two managers are the only ones currently supporting all features that the wormhole system needs.

3.2.4 MONSN

One of the goals of the work is to make usage of MONSN networking system as easy as possible. [21] This is achieved through the implementation of a telepathy connection manager using the MONSN network.

MONSN is very generic networking system, basing itself on one of the original ideas of the Internet: each and every node can access each and every other node in the network. This is achieved through tunnels to a server that assigns each node a public, reachable IP address. It also add methods for controlling incoming traffic on network level, which is needed on mobile pay-by-traffic subscriptions.

MONSN client software also implements a local firewall, and incoming connections must be started with the MONSN protocol. The protocol verifies the identity of the connecting party, and the client software then opens ports, if the incoming request is accepted.

MONSN does not specify application protocols to be used. Instead, normal Internet protocols should be used. TCPv6 and UDPv6 are supported, and the Telepathy connection manager needs to implement the application level protocol for instant messaging, calls and file transfer and stream tubes.

3.3 Contact storage

The actual contact storage for the platform is an open issue at this point. It will most likely be semantic storage.

Semantic storage is somewhat unlike traditional content storage systems. It provides many different kinds of relations between people. This allows for a contact chooser, that is not a traditional list, but rather a map, or a graph, in which the contacts are linked using named connections. The semantic contact storage is queried using SparQL query language. [18]

Advantages of a semantic backend is that it is designed to be extendable. Any information of the contacts can be trivially added to the database. While in this scope it is not needed, it may prove to be useful in further refinements of the system.

4. METHODS

The design goal of the API was to be as easy to use as possible. That goal is best achieved by streamlining the API for a selected set of functionalities, and leaving everything unnecessary out. Having both the device to device, and program to program functionalities was making the API too complex, and the program to program functionality was removed.

As the address book for the platform has not been decided on, the contacts are only referenced via string identifiers in the API. The string identifiers are platform dependent, and must be the same as used by the contact address book used.

Proper API usability assessment would require evaluating the API with examples and testing it on programmers, but in the scope of this study, the usability is measured by lines of code in certain use-cases.

A study on API learnability [19] suggests that the biggest obstacle for learning is poor documentation, especially missing examples. The second biggest obstacle is the structure of the API. The documentation should have rationale for the design, so the programmers would understand the design choices taken, and also include examples for most of the functionality.

4.1 Use cases

There are four major use-cases for the API: sending files and streams, and receiving them. All incoming streams and file transfers are pre-approved by the manager; so the API will only give notifications about them using Qt signals. There are also signals about the state of a file transfer. The notifications on streams are handled via normal QIODevice-controls.

Image viewer – sending file

A user is browsing through their collection of images. As he stumbles upon an image from a holiday, it reminds him of a friend that he was travelling with. He decides to send the image to the friend to share the memory. By tapping the toolbar item “Share” he initiates the transfer and a contact selector appears on the screen. Quickly browsing through the network of contacts, he finds the friend and taps on him to confirm. The picture transfer begins.

The use case is illustrated in Figures 4.1, 4.2 and 4.3 as mock-ups. The first

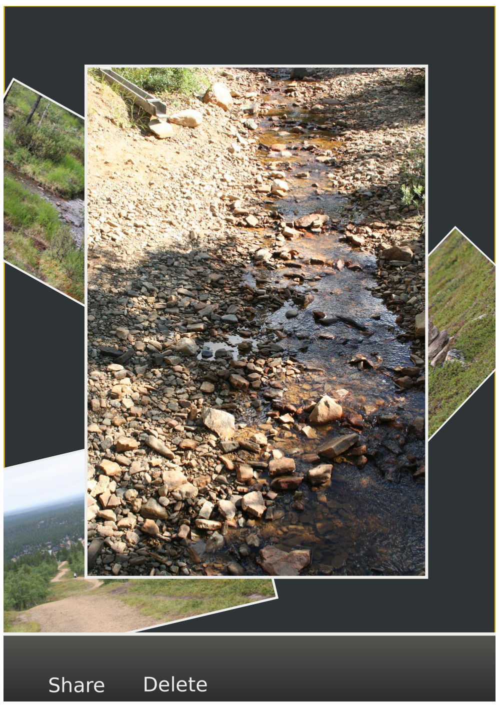


Figure 4.1: A screenshot mock-up of an image viewer

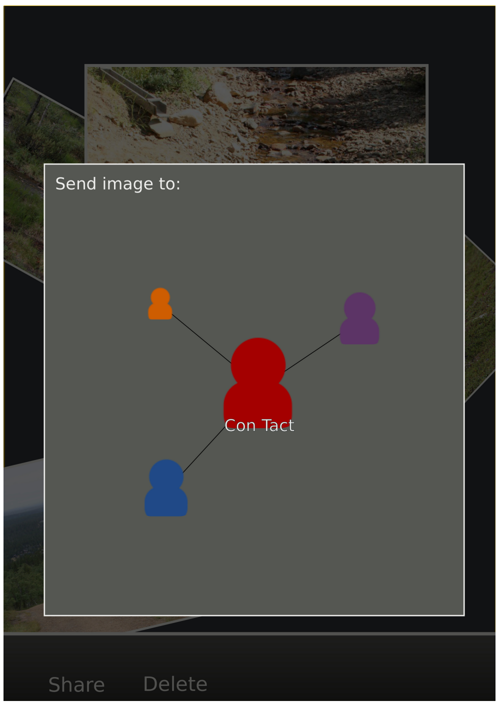


Figure 4.2: A screenshot mock-up of an image viewer contact selection

figure shows the image browser with the wanted picture visible. The toolbar at the bottom of the screen has a “Share” item. After tapping the item, the contact selector is shown, like in the second figure. The selector can be used to browse through the contact network. By tapping a contact at the edge, the contact is moved to the center. Choosing a contact is achieved by tapping the contact at the center. After choosing the contact the application returns to the main view with a progress bar in the toolbar, as shown in the third figure.

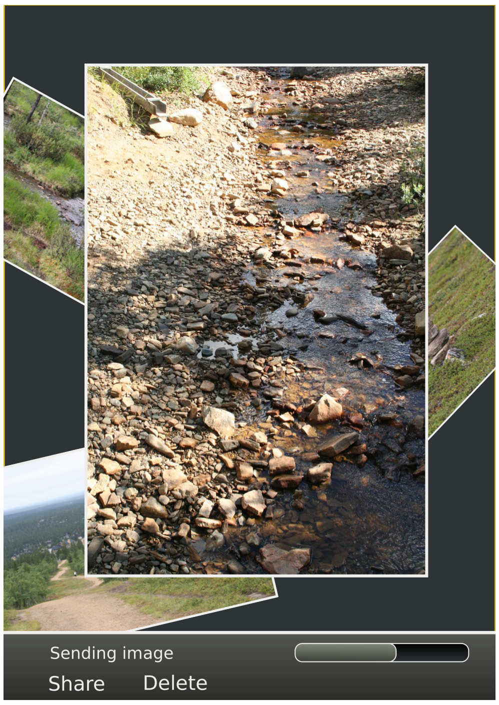


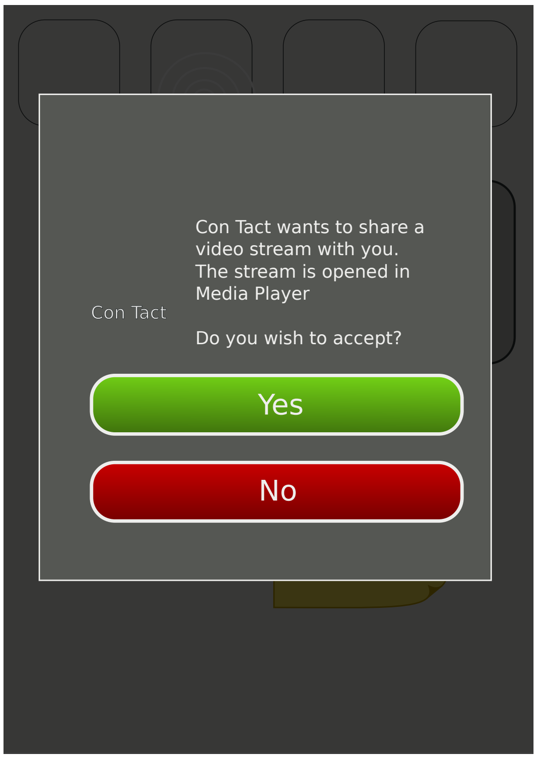
Figure 4.3: A screenshot mock-up of an image viewer sending a file

Media player – receiving stream

A friend has a cat that is acting funny. He figures that someone else might enjoy the display, and decides to share a video stream of the cat. When he chooses a share stream option in his video recorder and chooses the contact, the receiving end shows a notification. As the friend is up for a good laugh, he accepts the stream and the video player is launched showing the stream just as the cat runs into the dog.

The dialog that confirms the launch of the media player is shown in Figure 4.4. The dialog contains the name and picture of the contact that is sending the stream. It also states the program that will be used to handle the stream.

After confirming the request media player is launched. The media player automatically starts showing the stream, as seen in Figure 4.5. The user may stop the stream at any time by pressing stop or by closing the window.



Con Tact

Con Tact wants to share a video stream with you.
The stream is opened in Media Player

Do you wish to accept?

Figure 4.4: A screenshot mock-up of incoming stream

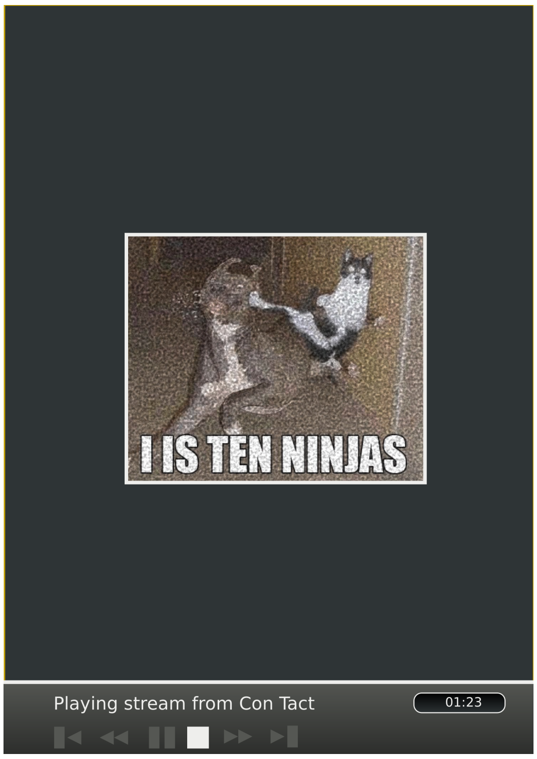


Figure 4.5: A screenshot mock-up of media player playing a video stream

4.2 Architecture

The implementation consists of two components, the Wormhole manager, that talks to the Telepathy framework and handles incoming requests; and the Wormhole client library, providing the API. The high level architecture is shown in Figure 4.6.

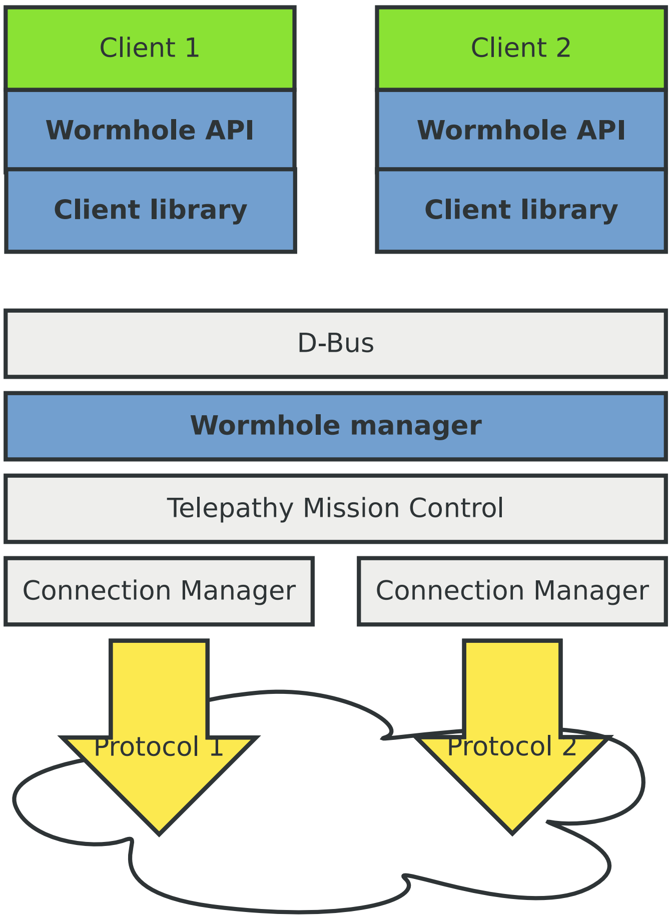


Figure 4.6: Architecture of the system, components with bold names and blue background are to be implemented

4.2.1 Manager

The Wormhole manager is a daemon-like entity, automatically started upon request using D-Bus autostart functionality. The manager exports objects over D-Bus, that represent the wormholes, the wormhole services and wormhole transfers. When the manager no longer has clients, it should terminate.

The D-Bus interfaces closely resemble the programming API. All additional functionality, like querying the contact from the user, is done in the manager process.

The manager class keeps track of clients that request wormholes, and closes all wormholes and associated transfers of a client if one dies. By doing this it prevents resource leaking. Otherwise all wormholes created by a client would live indefinitely if the client exits uncleanly.

The manager is responsible for querying the user for contact when a wormhole is created without a target. It must also ask the user when a file or stream is received and a verification is needed for certain handler program.

The manager is responsible for resolving an action for the data being transferred. When sending it means choosing the correct protocol and transfer mode. When receiving, the manager needs to decide the correct application to handle the incoming data using the data type and possibly some other details.

4.2.2 Client library

The client library implements the Wormhole API. It acts as a simplifying layer between the D-Bus API of the manager and the client program. It takes care of proxy object creation for objects exported by the manager, and hides all details of the D-Bus usage.

The client library uses a singleton object that keeps track of all objects created by and for the application. Upon destroying them the object notifies the manager over D-Bus, so the manager knows that the resource is no longer needed. It also communicates with the manager to request new wormholes and to register new wormhole enabled service.

The client library uses QSharedPointers in the implementation of the API and QSharedPointers and QWeakPointers internally. QWeakPointers are used to reference objects that should be destroyed when the application no longer needs them.

4.3 Security considerations

Incoming files may launch almost any program, and even cause installation of one, so the trustworthiness of the source needs to be properly evaluated. The evaluation can be done using three factors: how well is the sender known, how well does the protocol idenfity the sender and what is the potential impact of the action.

Security is most important for file types that contain program code, that is run automatically. This type of action is planned at least for JavaScript files. For executable files, the wormhole system should require strongly trusted sender or query from the user.

Some protocols provide high level of identification through cryptographic certificates. Currently only protocol that provides this is MONSN. In other protocols, the trustworthiness is that of the server relaying the messages.

In Table 4.1 we describe how a level of trust can be approximated from the trust level of the contact and that of the protocol. When protocol provides some level of trust, even unknown contacts can have some trust.

Table 4.1: Table of trust levels

		Protocol provided trust		
		strong	weak	none
User trust	friend	5	4	3
	acquaintance	4	3	2
	unknown	3	2	1

In the table, strong protocol trust refers to cryptographic identification. The cryptography is not enough by itself, but the certificate used must be signed by some trusted party. For centralized networks this means that the network operator signs the certificates. Even completely distributed peer-to-peer protocols may provide strong trust via a certificate web of trust as used in with PGP [2]. While in PGP web of trust keyservers are used to find a path from a trusted party to the certificate being evaluated, distributed systems have no keyserver, and all the middle certificates need to be known.

None refers to protocols where any user may claim to be anyone. An example of such network is the link local XMPP, where any node on the local network may claim a name, and no other details are provided. IRC [8] can also be considered to provide no trust, if only the nickname is known.

All protocols should be carefully evaluated on a much finer scale. The protocol trust level can also be dynamic; for example depending on the length of the certificate path.

The user trust level could be calculated from the semantic contact storage. Possible factors could be amount of common friends or type of relation. A spouse and other immediate family could be ultimately trusted, and others scaled down from that. As with protocol, the contact trust should also be much more granular.

For further improvements in the security, the data should also be passed to the application. This way received scripts could be run with different privileges depending on the identification factors.

4.4 Configuration

The system needs to know what program some type of file or stream should be passed to. The files are identified by mime types [3], and a mapping from a mime type to a program is needed. Normally, on GNU/Linux systems, this is done using .desktop files. These are plain text files containing key-value pairs, with certain well known, pre-defined keys for certain uses, but the set of keys can be extended. The simples solution is to add an extension field to these files for identifying the wormhole-aware programs.

For further configuration, additional metadata should be added. In the scope of this thesis, all wormhole programs are treated exactly the same. The program should be able to state the risk factor of running it, so this data should be added.

The user should be able to control what kind of actions are run automatically, what are confirmed from the user, and what are rejected. This data can not live in the .desktop files, as they are not user files, so another data storage is needed. This decision is not made at this point, as no user tuning is implemented.

5. IMPLEMENTATION

As the result of the interface design, the wormhole API was created. The wormhole API enables an application developer to use the wormhole system. The API is provided by the client library, which is a thin wrapper on top of the D-Bus API provided by the wormhole manager. The D-Bus API can also be used by itself, should the developer require higher control of the application flow during the calls to API methods.

5.1 Program-level interface

The overall design of the program-level interface is shown in Figure 5.1. The interface consist of the three classes, Wormhole, WormholeServer and WormholeFile in the Client library package.

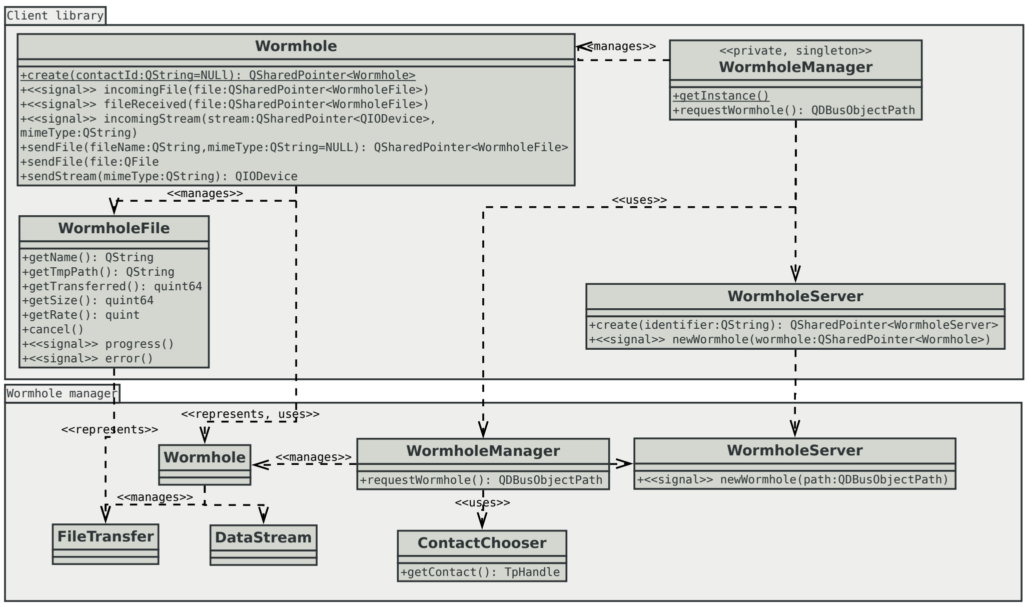
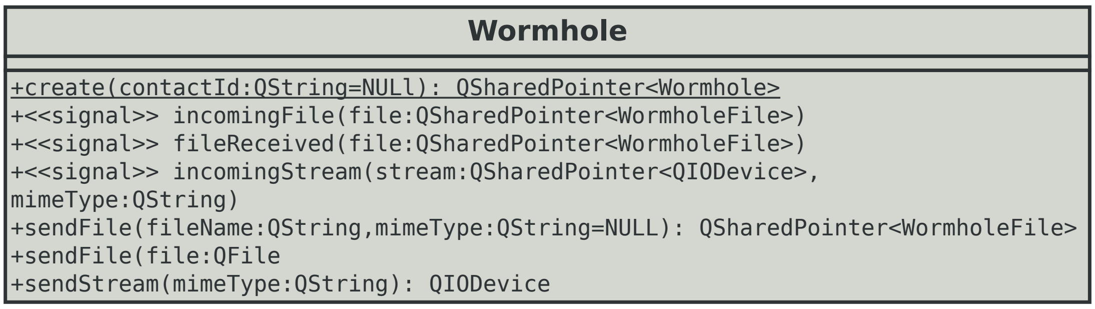

Client library
Wormhole
+create(contactId:QString=NULL): QSharedPointer<Wormhole>
+<<signal>> incomingFile(file:QSharedPointer<WormholeFile>)
+<<signal>> fileReceived(file:QSharedPointer<WormholeFile>)
+<<signal>> incomingStream(stream:QSharedPointer<QIODevice>, mimeType:QString)
+sendFile(fileName:QString,mimeType:QString=NULL): QSharedPointer<WormholeFile>
+sendFile(file:QFile
+sendStream(mimeType:QString): QIODevice
<<manages>>
<<private, singleton>>
WormholeManager
+getInstance()
+requestWormhole(): QDBusObjectPath
<<manages>>
<<uses>>
WormholeFile
+getName(): QString
+getTmpPath(): QString
+getTransferred(): quint64
+getSize(): quint64
+getRate(): quint
+cancel()
+<<signal>> progress()
+<<signal>> error()
WormholeServer
+create(identifier:QString): QSharedPointer<WormholeServer>
+<<signal>> newWormhole(wormhole:QSharedPointer<Wormhole>)
Wormhole manager
<<represents, uses>>
<<represents>>
Wormhole
<<manages>>
WormholeManager
+requestWormhole(): QDBusObjectPath
WormholeServer
+<<signal>> newWormhole(path:QDBusObjectPath)
<<manages>>
<<uses>>
FileTransfer
DataStream
ContactChooser
+getContact(): TpHandle

Figure 5.1: Class diagram of the architecture of the system.

5.1.1 Wormhole class

The wormhole class is the main class of the API. The class is used for sending files and streams and receiving them. Only wormholes received from a WormholeServer



```
Wormhole

+create(contactId:QString=NULl): QSharedPointer<Wormhole>
+<<signal>> incomingFile(file:QSharedPointer<WormholeFile>)
+<<signal>> fileReceived(file:QSharedPointer<WormholeFile>)
+<<signal>> incomingStream(stream:QSharedPointer<QIODevice>,
mimeType:QString)
+sendFile(fileName:QString,mimeType:QString=NULL): QSharedPointer<WormholeFile>
+sendFile(file:QFile
+sendStream(mimeType:QString): QIODevice
```

Figure 5.2: Details of the Wormhole class

signal can receive. The methods and signals are shown in Figure 5.2.

The API consists of one static method for creating wormholes, three signals for incoming files and streams and three methods for sending files and streams.

create method

Wormhole is an abstract class, and cannot be instantiated with the new keyword. It can only be instantiated via the create() method, which uses the WormholeManager class to request a wormhole object. The method is described in Table 5.1. The ownership of the returned object should be handled using shared pointers. Successive calls to the create() method may return the same object, if the user selects the same contact and the previous one was not destroyed.

Table 5.1: Arguments and return value of the create() method

create()	
Arguments	1
contactId	(Optional) A string identifier of the contact to which the wormhole will be connected to. If not specified or NULL, the manager will query contact from the user.
Return value	A shared pointer to the wormhole object, or NULL pointer if the user did not select a contact, or the given contact identifier was not valid.
Errors	May fail if no connection to D-Bus, not enough memory, or some internal error in manager. In error cases returns NULL pointer.
Notes	The Qt main loop will be executed during the call to this method.

If the optional contactId argument is given, the wormhole is connected to the contact identified by the identifier. Support for this is included for use cases, where the application wants to handle contact selection itself, like in the home screen scenario (Figure 2.2, page 4).

sendFile method

The sendFile() method prepares for sending the passed file to the other end of the wormhole. If passed, the mime-type is used to determine the proper action for the file. The method has two versions, one that takes in a QFile reference (Table 5.2), and another that takes in a file name (Table 5.3). The former will resolve the file name and call the latter.

Table 5.2: Arguments and return value of the sendFile() method

sendFile()	
Arguments	2
file	A reference to an open QFile. The ownership of the QFile is moved to the returned WormholeFile object, and will be closed once the object is destroyed.
mimeType	(Optional) A string specifying the mime-type of the sent file. If not specified, determined from the file name returned by the fileName() method of passed file.
Return value	A shared pointer to the wormhole file object, or NULL pointer on error.
Errors	If the file is not open, or the manager could not determine transfer channel for the file, returns NULL pointer.

Table 5.3: Arguments and return value of the overloaded sendFile() method

sendFile()	
Arguments	2
fileName	The name of the file to be sent. May be relative to current working directory, or a absolute path.
mimeType	As in 5.2.
Return value	A shared pointer to the wormhole file object, or NULL pointer on error.
Errors	If the file could not be opened or the manager could not determine transfer channel for the file, returns NULL pointer.
Notes	The Qt main loop may be executed during the call to this method.

The method will seek the passed file to the beginning before sending. The file will be transferred even if the caller drops its reference to the WormholeFile and Wormhole objects. Only way to cancel the transfer is by using the cancel() method of the WormholeFile (see 5.1.2).

The file sent must be a regular file, and its size must be known at the time the transfer starts, and it must not be changed during the transfer.

sendStream method

The sendStream() method will open a stream to the other end of the wormhole and return a QIODevice object. The method is described in Table 5.4. All data written into the QIODevice is passed through the stream.

Table 5.4: Arguments and return value of the sendFile() method

sendStream()	
Arguments	1
mimeType	A mime type for the data being sent. Must be defined to determine correct application at receiving end.
Return value	A QIODevice that can be used to transfer data. On errors a closed QIODevice is returned.
Errors	If the manager could not determine transfer channel for the stream, a closed QIODevice is returned.
Notes	The QIODevice refers to a QIODevice object owned by the Wormhole, and all streams are closed when the Wormhole object is destroyed.

For easier handling, the QIODevice may be used through the Qt wrappers like QDataStream.

incomingFile signal

The incomingFile signal is emitted whenever a file transfer request is received from the other end of the wormhole. The signal is described in Table 5.5. The signal is only emitted by the wormholes that are created by a WormholeServer object that is handling the mime type of the file.

Table 5.5: Parameters of the incomingFile signals

incomingFile	
Parameters	1
file	A shared pointer to object representing the file being received.

If the pointer is not shared, the transfer will be continued, and fileReceived signal is emitted once the transfer is complete. The transfer can be cancelled at any point using the cancel() method of the WormholeFile (see 5.1.2).

receivedFile signal

The receivedFile signal is emitted whenever a file transfer is completed. The signal is described in Table 5.6. The signal is only emitted by the wormholes that are created by a WormholeServer object that is handling the mime type of the file.

Table 5.6: Parameters of the receivedFile signal

receivedFile	
Parameters	1
file	A shared pointer to object representing the received file.

If the pointer is not shared, the WormholeFile is destroyed and the file will be deleted. If the file should persist, it should be moved to another place before releasing the WormholeFile.

Code example

In Program 5.1 is a listing of choosing a file to send and then requesting a Wormhole to send the file to. The code first creates a file open dialog, requesting the user to select a file. It then creates a Wormhole, which causes the Wormhole manager to query a contact from the user. Then it calls the sendFile method, which is shown in Program 5.2.

```
void SomeClass::beginSending()
{
  QString file = QFileDialog::getOpenFileName();

  if (file.isNull())
  {
    // User pressed cancel.
    return;
  }

  this.wormhole = Wormhole::create();
  if (this.wormhole.isNull())
  {
    // User pressed cancel.
    return;
  }
  sendFile(file);
}
```

Program 5.1: Simple code example of usage of Wormhole class

5.1.2 WormholeFile class

The WormholeFile class is used for following the progress of file transfers. It can be queried for details of the file, and used to cancel the transfer. The methods and signals are shown in Figure 5.3.

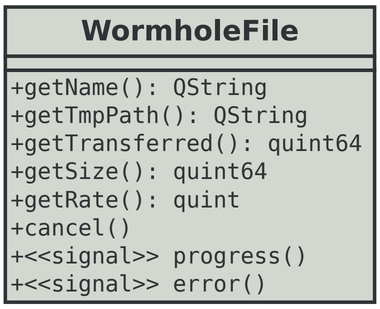


Figure 5.3: Details of the WormholeFile class

The wormhole will keep a strong reference (a QSharedPointer) to the object until the transfer is complete or aborted. After emitting the fileReceived or error signal, Wormhole drops the reference, and if no one else has a reference, the WormholeFile is destroyed.

When a WormholeFile is destroyed, the temporary file is unlinked. Before dropping its reference on the object, the user should move the file to another location. Special care should be taken when using the object from multiple locations, because only one should move the file. On platforms that support it, hard links may be used to fight this problem, but they should only be used if really needed.

getName method

The getName() method returns the file name reported by the sending end. The name can be used to determine the file name onto which the transferred file is saved, or as a hint in a "Save as" dialog. The getName() method is described in Table 5.7.

The returned name is just as reported by the sending end, the extension used might not match the mime type of the file. When determining the action on the file, the mime type should be used. Only if the mime type is reported as "application/octet-stream", should the file name be taken into account.

getTmpPath method

The getTmpPath() method returns the temporary file used for saving the file being transferred. The getTmpPath() method is described in Table 5.8.

On destruction of the object, the returned file is unlinked.

Table 5.7: Arguments and return value of the getName() method

getName()	
Arguments	None.
Return value	A string containing the name of the file being transferred. This is just the file name reported by the sending end, no path information.
Errors	If the sending end did not report a name, may return NULL.
Notes	The file extension of the file should not be used to determine the action, but rather the mime type.

Table 5.8: Arguments and return value of the getTmpPath() method

getTmpPath()	
Arguments	None.
Return value	A string containing the temporary name of the file being transferred with full path. When transfer is complete, the user should move the file.
Errors	Returns NULL for files being sent.
Notes	The returned file name may not have correct file extension, for naming purposes use the getName().

getTransferred method

The getTransferred() method returns the amount of bytes that have already been transferred. The getTransferred() method is described in Table 5.9.

Table 5.9: Arguments and return value of the getTransferred() method

getTransferred()	
Arguments	None.
Return value	An integer containing the amount of bytes already transferred of the file.
Errors	None.
Notes	The returned value may be lower than the actual amount of data. The value returned is from the same time as the previous progress signal.

A call to the getTransferred() method does not cause a round-trip over D-Bus, but rather returns the local value received from the daemon process in last progress update. More accurate figure might be obtainable using the stat() system function on the temporary file.

If a percentage number is needed, it can be obtained using the following snippet:

```
qdouble percentage = file.getTransferred() * 100 / file.getSize();
```

getSize method

The getSize() method returns the total amount of bytes in the file being transferred. The getSize() method is described in Table 5.10.

Table 5.10: Arguments and return value of the getSize() method

	getProgress()
Arguments	None.
Return value	An integer containing the total bytes of the file.
Errors	None.

The value returned by getSize is always valid. If the size of the file changes during transfer, the transfer is cancelled once noticed.

getRate method

The getRate() method returns the average transfer rate for the last ten seconds. The value is given in bytes per second. The getRate() method is described in Table 5.11.

Table 5.11: Arguments and return value of the getRate() method

	getRate()
Arguments	None.
Return value	An integer containing the average number of bytes transferred in a second over the last 10 seconds.
Errors	None.
Notes	The returned value may differ from the current transfer rate. The value returned is from the same time as the previous progress signal.

The remaining transfer time can be calculated with following code:

```
qdouble seconds = (file.getSize() - file.getTransferred()) /
                  file.getRate();
```

cancel method

The cancel() method will cancel a file transfer. Received data is no explicitly destroyed, the cancelled object will stay usable. Calling cancel() will cause the error signal to be emitted, and getRate() method will return 0 for subsequent calls. The cancel() method is described in Table 5.12.

Table 5.12: Arguments and return value of the cancel() method

cancel()	
Arguments	None.
Return value	None.
Errors	None.
Notes	It is safe to call the function multiple times. Any subsequent calls are silently ignored.

Once a file transfer is cancelled, the connection is closed, and no data is added to the file anymore. The getTransferred() will return the actual amount of bytes that were transferred up to the cancellation point.

progress signal

The progress signal is emitted periodically during a file transfer. The signal is described in Table 5.13. When the WormholeFile receives a notification from the daemon, it will update the transferred byte count and transfer rate and emit progress signal.

Table 5.13: Parameters of the progress signal

progress	
Parameters	None.

The delay between progress signal emissions is the minimum of the following three values:

- 10 seconds,
- duration of 1% of total transfer, but minimum of 5 seconds and
- 1 second, if remaining transfer time is less than 10 seconds.

These rules should allow decent interactivity for the user without flooding the D-Bus message bus.

error signal

The error signal is emitted when a file transfer is stopped for some reason. The signal is described in Table 5.14.

Table 5.14: Parameters of the error signal

	error
Parameters	None.

Code example

In Program 5.2 is a simple example of sending a file and following the progress until the transfer is complete.

```
void SomeClass::onProgress()
{
  quint64 total = file->getSize();
  quint64 bytesTransferred = file->getTransferred();

  if (bytesTransferred == total)
  {
    qDebug() << "Transfer complete.";
    file = NULL;
  }
}

void SomeClass::sendFile(QString &file)
{
  QSharedPointer<WormholeFile> file = this.wormhole.SendFile(file);

  if (file.isNull())
  {
    // Something failed.
    return;
  }

  this.file = file;

  connect(file.data(), SIGNAL(progress()),
          this, SLOT(onProgress()));
}
```

Program 5.2: Simple code example of usage of WormholeFile class

5.1.3 WormholeServer class

The server class is for receiving incoming wormhole requests. All wormholes coming from the server object are validated by the manage and no user interaction is needed.

The methods and signals are shown in Figure 5.4.

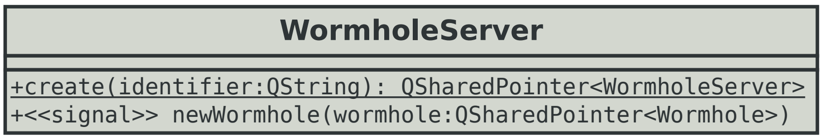

WormholeServer
+create(identifier:QString): QSharedPointer<WormholeServer>
+<<signal>> newWormhole(wormhole:QSharedPointer<Wormhole>)

Figure 5.4: Details of the WormholeServer class

Once a WormholeServer class is instantiated with the create() method, the signal must be connected to before allowing the Qt main loop be run. The object will register itself with the daemon, and may immediately receive incoming wormhole notifications when entering the Qt main loop.

create method

WormholeServer is an abstract class, and cannot be instantiated with the new keyword. It can only be instantiated via the create() method, which uses the WormholeManager class to request a wormhole server object. The method is described in Table 5.15. The ownership of the returned object must be handled using shared pointers. Successive calls to the create() method may return the same object, if the identifier is the same.

Table 5.15: Arguments and return value of the create() method

create()	
Arguments	1
identifier	A string identifier for the program being run. The identifier must be the same as the .desktop file name (see 4.4).
Return value	A shared pointer to the wormholeserver object, or NULL pointer if the program identifier was not valid.
Errors	May fail if no connection to D-Bus, not enough memory, or some internal error in manager. In error cases returns NULL pointer.
Notes	The Qt main loop will be executed during the call to this method.

newWormhole signal

The newWormhole signal is emitted whenever a request is received from a contact to the program. The signal is described in Table 5.16.

If the pointer is not shared, all transfer are cancelled, including file transfers. The receiver should connect to all signals of the wormhole object before returning.

Table 5.16: Parameters of the newWormhole signal

newWormhole	
Parameters	1
wormhole	A shared pointer to an incoming wormhole object.

Code example

In Program 5.3 is an example of creating a WormholeServer and reading all data from an incoming stream. The program first creates the WormholeServer instance and connects the newWormhole signal to the gotWormhole handler. Whenever the signal is received, the handler connects the incomingStream handler to the incomingStream signal of the new Wormhole got as the argument of the signal.

5.2 D-Bus Interfaces

The interfaces for the communication between client and manager are important. The different interfaces are described in the following subsections. In the tables describing the D-Bus methods and signals, the signature and return values fields contain the D-Bus signature. The signature contains the type identifiers of all arguments joined together. For example, signature “ss” means two string arguments. The used data types are shown in Table 5.17.

The user of any of the methods in any of the D-Bus interfaces should prepare for the D-Bus set of standard errors [4] in addition to the ones mentioned in descriptions. These standard errors are used where appropriate. They are described in method details, except for insufficient memory and no network situations, these standard errors are common for all methods.

Table 5.17: D-Bus datatypes

Identifier	Data type
s	String
o	Object path
t	64-bit unsigned integer
u	32-bit unsigned integer

5.2.1 com.nokia.Wormhole.Manager interface

The com.nokia.Wormhole.Manager interface is used to register clients. The interface is implemented by a manager object, that is exported to D-Bus object path “/”.

```
MyClass::MyClass()
{
  ...
  this.wormholeServer = WormholeServer::create("MyApplication");
  connect(this.wormholeServer.data(),
          SIGNAL(newWormhole(QSharedPointer<Wormhole>)),
          this,
  SLOT(gotWormhole(QSharedPointer<Wormhole>)));
}

void MyClass::gotWormhole(QSharedPointer<Wormhole> wormhole)
{
  this.wormhole = wormhole;
  connect(wormhole.data(),
          SIGNAL(incomingStream(QSharedPointer<QIODevice>)),
          this,
  SLOT(incomingStream(QSharedPointer<QIODevice>)));
}

void MyClass::incomingStream(QSharedPointer<QIODevice> stream)
{
  this.stream = stream;

  connect(stream.data(), readyRead(),
          this, incomingData());
}

void MyClass::incomingData()
{
  QByteArray data = this.stream->readAll();
  qDebug() << "Read " << data.size() < " bytes of data.";
}
```

Program 5.3: Simple code example of usage of WormholeServer class

The interface has two methods. One for requesting a wormhole object, and one for requesting a wormholeserver object. The details of the former are in Table 5.18 and the latter in Table 5.19.

5.2.2 com.nokia.Wormhole.Wormhole

The com.nokia.Wormhole.Wormhole interface is implemented by the Wormhole objects in the manager. Any object returned by RequestWormhole in the com.nokia. Wormhole.Manager, or received in a NewWormhole signal in the com.nokia. Worm-

Table 5.18: Details of the RequestWormhole method

RequestWormhole	
Description	The RequestWormhole method creates a Wormhole object to the contact identified by contact argument, or to a contact queried from the user.
Signature	s
contact	A string identifier of the contact to which the wormhole is requested. Empty string if not known.
Return values	o
path	The object path of the created Wormhole object.
Errors	If the end user does not select any contact, or the provided contact does not match any in the, returns error "com.nokia.Wormhole.Error.NoContact."

Table 5.19: Details of the RegisterServer method

RegisterServer	
Description	The RegisterServer creates a WormholeServer object for the client application identified by identifier argument.
Signature	s
identifier	A string identifier for the program being run. The identifier must be the same as the .desktop file name (see 4.4).
Return values	o
path	The object path of the created WormholeServer object.
Errors	If the provided identifier cannot be used to identify a program, returns error "com.nokia.Wormhole.Error.NoSuchClient."

hole.Server must implement this interface.

The Interface has two methods: one for sending a file and one for starting an outgoing stream. The methods are described in Tables 5.20 and 5.21 respectively. It also has two signals: one for notifying on incoming files, and one for notifying on incoming streams. The signals are described in Tables 5.22 and 5.23 respectively.

5.2.3 com.nokia.Wormhole.Server

The com.nokia.Wormhole.Server interface provides the signalling for incomng wormholes. The interface only has one signal, the NewWormhole signal. The details of the signal can be found in Table 5.24.

Table 5.20: Details of the SendFile method

SendFile

Description	The SendFile method begins transfer of the file given in the path argument.
Signature	sss
path	The absolute file path at which the file can be found; must be a local file.
filename	The name for the file that is reported to the recipient.
mimeType	The mime type of the file being sent.
Return values	o
path	The object path of the created WormholeFile object.
Errors	If no file transfer to the contact of the Wormhole object can be made, returns error "com.nokia.Wormhole.Error.NoRoute."

Table 5.21: Details of the SendStream method

SendStream

Description	The SendStream method opens a stream over the wormhole passing through any data written into the named pipe.
Signature	ss
path	The file path of the named pipe.
mimeType	The mime type of the stream being sent.
Return values	o
path	The object path of the created WormholeStream object.
Errors	If no stream transfer to the contact of the Wormhole object can be made, returns error "com.nokia.Wormhole.Error.NoRoute."

Table 5.22: Details of the IncomingFile signal

IncomingFile

Description	The IncomingFile signal is sent whenever there is an incoming file request from the contact.
Signature	o
path	The object path of the WormholeFile object created for the incoming file.

5.2.4 com.nokia.Wormhole.File

The com.nokia.Wormhole.File interface is implemented by objects at paths returned by the SendFile in com.nokia.Wormhole.Wormhole, or those received as parameter for IncomingFile signal from the same interface. The interface has one method and

Table 5.23: Details of the IncomingStream signal

IncomingStream	
Description	The IncomingStream signal is sent whenever there is an incoming stream request from the contact.
Signature	o
path	The object path of the WormholeStream object created for the incoming stream.

Table 5.24: Details of the NewWormhole signal

NewWormhole	
Description	The NewWormhole signal is sent whenever a request for a file or stream is received and there was no Wormhole instance to handle the request.
Signature	o
path	The object path of the Wormhole object created for the incoming wormhole.

one signal.

The GetDetails method is described in Table 5.27 and the Progress signal in Table 5.26. The Progress signal is emitted at intervals as described in progress signal under subsection 5.1.2.

Table 5.25: Details of the GetDetails method

GetDetails	
Description	The GetDetails method fetches information of the file transfer.
Signature	
Return values	ssstt
filename	The name for the file being transferred.
path	The name of the file being sent, or the temporary path where the file is being saved.
mimeType	The mime type of the file being transferred.
size	The size of the file being transferred.
transferred	The amount of bytes already transferred.
Errors	Common errors apply.

5.2.5 com.nokia.Wormhole.Stream

The com.nokia.Wormhole.Stream interface is implemented by objects at paths returned by the SendStream method in the com.nokia.Wormhole.Wormhole interface,

Table 5.26: Details of the Progress signal

	Progress
Description	The Progress signal notifies on updates on the transfer.
Signature	tu
transferred	The amount of bytes already transferred.
rate	The transfer rate in bytes per second.

or those received as parameter for IncomingStream signal on the same interface. The Stream interface only has one method for getting basic details of the stream. Actual data transfer is done via the named pipe.

Table 5.27: Details of the GetDetails method

	GetDetails
Description	The GetDetails method fetches information of the stream.
Signature	
Return values	ss
path	The file path of the named pipe.
mimeType	The mime type of the file being transferred.
Errors	Common errors apply.

5.2.6 com.nokia.Wormhole.Object

All wormhole objects implement the com.nokia.Wormhole.Object interface. The interface has only one method, UnRef. Calling the method will drop the reference of the calling service to the object. All remote references are named, so calling this method more than once from one service has no effect. The client library should always call the UnRef method when a representing instance is destroyed.

Table 5.28: Details of the UnRef method

	GetDetails
Description	Drops the callers reference to the object.
Signature	
Return values	

5.3 Implementation of the client library

The client library is mostly a thin convenience wrapper for the D-Bus API of the manager. It hides complexity of asynchronous calls from the application and also all the trouble of proxy object creation from the returned object paths.

The WormholeManager class connects to the D-Bus session bus upon creation and the same connection is used for all created objects. Each remote object class has an autogenerated interface class that is a subclass of QDBusAbstractInterface. The interface classes are used to do the D-Bus calls.

The library contains the abstract public classes introduced in the API section, and also the private actual implementations, that inherit the public ones. This is to further hide any implementation details from the programmer, they only see things they need to care about. Each public class, Wormhole, WormholeFile and WormholeServer has a private counterpart, WormholeImpl, WormholeFileImpl and WormholeServerImpl. The main reason for the need of these is that in Qt signals can only be listened to by QObjects, and signals can only be emitted by the object itself.

For streams, the library creates a named pipe, and the name of the pipe is passed to the manager. The returned QIODevice object points to this pipe. This way, the data will be passed straight through from the client to the manager, without having D-Bus in between; it will save a process switch, which is important on mobile devices.

For files, the library will pass the file name with full path. The file must be accessible using the standard QFile class.

5.4 Implementation of the manager

The manager is implemented in the standard D-Bus manner. The actual manager object is exported into D-Bus object path "/". Upon request, the manager creates wormhole objects and server objects.

Each wormhole instance is generated a unique object path, which is then passed to the client that requested the instance. The path starts with "/wormhole/", then the contact id stripped of disallowed characters (only alphanumerics and underscore are allowed), and finally a running number. Even though it would be possible to use a shared instance for all requests for sending, each request will be served with a newly created instance. For servers the path should start with "/server/", and the name of the program is appended. This is enough to uniquely identify the server object, and no further data is needed.

When a new wormhole instance is created, the D-Bus service that requested it is monitored and if it goes away during the lifetime of the wormhole, the wormhole is destroyed, and all associated transfers are closed. The following is implemented using the NameOwnerChanged-signal from the D-Bus bus. The manager only listens for the unique names of the processes that have registered a wormhole. The WormholeServer instances are handled in a similar manner.

The architecture of the manager is as alread presented in Figure 5.1 on page

23. It contains counterparts for all the public classes in the API. In addition, it also contains a ContactChooser class, that is a dialog for choosing a contact, and a DataStream object, that is the counterpart for the streams of type QIODevice in the API.

6. CONCLUSIONS

The resulting API can be evaluated using the defined use-cases. For the image viewer to be able to send a file, it will need to

- request a wormhole using the Wormhole::create()-method,
- check the return value,
- send the file using the Wormhole::sendFile()-method,
- connect to two signals of the returned WormholeFile object and
- implement handler slots for the signals.

This sums up to about 10 lines of code, including the update of the progress meter in the toolbar.

In the other use-case, the video player, the player needs to

- register itself to receive data using the WormholeServer::create() method,
- connect to the newWormhole signal,
- implement handler slot for the newWormhole signal,
- register to the Wormhole::incomingStream-signal and
- implement handler slot for incomingStream signal.

This adds, again, about 10 lines of extra code. The incomingStream signal gives a standard QIODevice, and the player is expected to already know how to handle such.

All in all, the amount of code lines goes quite low. Mostly this is due to not having to choose the contact, or assess the trustworthiness of the incoming request. Even the Wormhole class could be skipped, but it might not be a good design decision as then the program would not have any pointer to the contact it is interacting with.

The Wormhole class could use a SendMessage() method for sending normal IM messages. This way the same interface could be used to quickly comment on transmitted data.

In the future, it might be useful to be able to query all ongoing transfers, and allow creating persistent file transfers. That way the system could have a single UI for all file transfers, and the applications could ignore all details of the progress. This would also allow sending files to contacts that are not available for file transfer during the lifetime of the application.

BIBLIOGRAPHY

[1] Collabora. telepathy-qt4 source. http://git.collabora.co.uk/?p=telepathy-qt4.git;a=summary. Read 22.4.2010.

[2] Patrick Feisthammel. Pgp: Explanation of the web of trust of pgp. http://www.rubin.ch/pgp/weboftrust.en.html. Read 21.5.2010.

[3] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. RFC 2046 (Draft Standard), November 1996. Updated by RFCs 2646, 3798, 5147.

[4] freedesktop.org. D-bus: Protocol constants. http://dbus.freedesktop.org/doc/api/html/group__DBusProtocol.html. Read 31.5.2010.

[5] freedesktop.org. freedesktop.org - Software/dbus. http://www.freedesktop.org/wiki/Software/dbus. Read 28.5.2010.

[6] freedesktop.org. Telepathy Wiki - Components. http://telepathy.freedesktop.org/wiki/Components. Read 28.5.2010.

[7] GNOME Foundation. Empathy - gnome live! http://live.gnome.org/Empathy. Read 2.6.2010.

[8] C. Kalt. Internet Relay Chat: Architecture. RFC 2810 (Informational), April 2000.

[9] Danielle Madeley and Murray Cumming. Telepathy Developer's Manual. http://people.collabora.co.uk/~danni/telepathy-book/. Read 22.4.2010.

[10] Nokia. Maemo Software | Nokia > The software behind your mobile computer. http://maemo.nokia.com/. Read 2.6.2010.

[11] Nokia. Qt - A cross-platform application and UI framework. http://qt.nokia.com/. Read 28.5.2010.

[12] Nokia. Qt 4.6 Events and Filters. http://doc.qt.nokia.com/4.6/eventsandfilters.html. Read 22.4.2010.

[13] Nokia. Qt 4.6: QSharedPointer Class Reference. http://doc.qt.nokia.com/4.6/qsharedpointer.html. Read 22.4.2010.

[14] Nokia. Qt 4.6 Signals and Slots. http://doc.qt.nokia.com/4.6/signalsandslots.html. Read 22.4.2010.

[15] One Laptop Per Child association. OLPC. `http://wiki.laptop.org/go/The_OLPC_Wiki`. Read 2.6.2010.

[16] Pidgin developer team. Pidgin, the universal chat client. `http://pidgin.im/`. Read 28.5.2010.

[17] Pidgin developer team. WhatIsLibpurple - Pidgin. `http://developer.pidgin.im/wiki/WhatIsLibpurple`. Read 28.5.2010.

[18] Eric Prud'hommeaux and Andy Seaborne. Sparql query language for rdf. `http://www.w3.org/TR/rdf-sparql-query/`. Read 22.4.2010.

[19] M.P. Robillard. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software*, 26(6):27–34, 2009.

[20] Bjarne Stroustrup. C++0x faq. `http://www2.research.att.com/~bs/C++0xFAQ.html`. Read 22.4.2010.

[21] Timo Strömmer. Tiedon siirto MONSN-verkossa. Master's thesis, University of Oulu, 2010.