Titol:	Estudio de la topología y protocolos de una red móvil ad-hoc
Volum:	1
Alumne:	Bruno Albalat Montenegro
Director/Ponent:	Josep Maria Barceló Ordinas
Departament	AC
Data:	23/11/2010

DADES DEL PROJECTE

Titol del projecte:	Estudio de la topología y protocolos de una red móvil ad-hoc.
Nom del estudiant:	Bruno Albalat Montenegro
Titulació:	Enginyeria en Informàtica Superior.
Crèdits:	37,5
Director/Ponent:	Josep Maria Barceló Ordinas
Departament	AC

Membres del tribunal (nom i signatura)

President:

Vocal:

Secretari:

Qualificació:

Qualificació numèrica:

Qualificació descriptiva:

Data:

Index

Introduction

Summary

The main topic of this FCP will be to present software that will be ease the task of testing Delay Tolerant Networks (henceforth DTNs) in a controlled environment. Therefore, a piece of software that will allow heavy testing using libraries that support ad-hoc networking will be developed, based in Nokia's Symbian® operative system.

Motivation

Although they have many proven uses, ad-hoc networks have never been really popular due to they're challenging implementation costs. There is, because of that, a somewhat big opportunity of investigation on these kinds of networks, and a huge potential to discover when working with them.

My main motivations for working in this project have been:

- The ability to investigate an "abandoned" theme: The fact that ah-hoc networking is quite the unknown field has proven to make it a severely interesting investigating topic, as every paper read and every conversation I had with my colleagues successfully made me learn tons of things about it.
- Opportunity to work with experienced researchers: Getting involved in this project has given me the opportunity to get personally involved in a research group that have a deserved reputation. The experience and know-how acquired from they're collaboration would be unachievable if working by myself.
- Connectivity as a motivation: Living in a "2.0" era, finding a project that involves new ways of connecting people, in a much more personal way than usual (due to physical proximity), is definitely a great motivation by itself. The vast possibilities that can derive of a project like this have been a great persuasion when considering project possibilities.
- Networking: Networking itself has always been an area that has specially interested me. Expanding my knowledge by researching on it is more than a pleasure.

Objective

The main objective of this project is to develop a delay tolerant messaging application that is going to use a custom mobile network.

Firstly, it would be interesting to describe what we mean by ‘delay tolerance’ in messaging applications: Our application will allow users to communicate between them sending

messages using only they're Wi-Fi (no text-messaging or any other carrier-dependant type of communication involved), tolerating a certain delay between the sending and the reception of the message. This is, we will allow users to send a message to other users that aren't connected to the network and expect them to receive them in a certain time window (providing the receiver connects to the network in this time window).

To do so, we are going to use a delay tolerant communication model based on Miraveo's® network libraries. These libraries will allow communication between mobile devices using only ad-hoc networks, and we will allow disconnection tolerance.

Basically, we are working with devices that won't always be present in the network. In fact, they will most probably be far more time disconnected that connected to the network. Furthermore, there won't be a single network, but single 'clouds' may appear, having completely different information in them, and ultimately having to merge this information. A typical use case would be that of two groups of friends that know each other but only meet occasionally.

Because of intermittency, we will have to be especially careful with message routing, and will have to give network users a unique ID that will ensure that messages directed to them will be delivered whenever possible if meeting the right conditions.

A point of interest of this application will be to make sure that the routing algorithm is easily located and modifiable, as in a future it can be of great use when deepening the research scope.

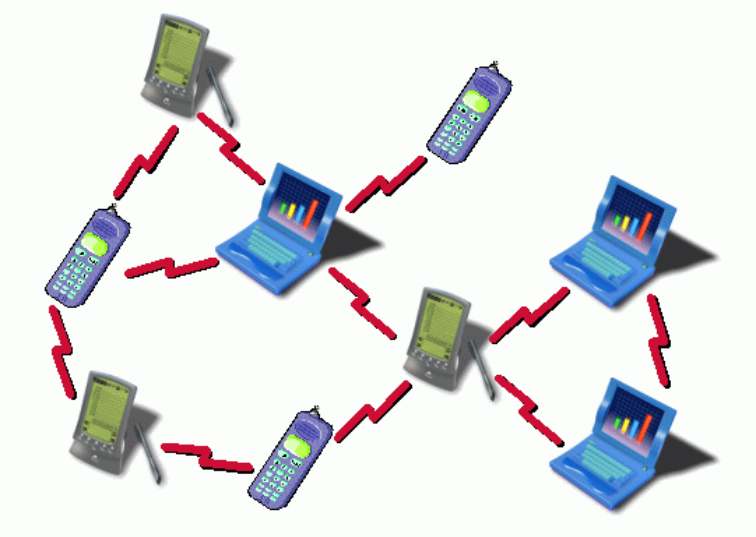
Summing up, the objective is to create a social application that will allow users to:

- Send messages to an individual or to a group: these messages can be either instant or delay-tolerant.
- Manage friends: To ease message sending, users will be able to create a unique Id that will ensure that they receive only the messages that are intended to them.
- Manage groups: In a similar fashion than when dealing with single users, we will also be able to create "interest groups" that will allow sending messages to a numbers of people that are part of that group.

This piece of software's purpose is mainly intended for investigation purposes, although any kind of approach can be given to it.

Ad-hoc networks

A mobile ad-hoc network (from here on MANET) simplest definition would be that of a self-configuring network of mobile devices connected by wireless links. These mobile devices can be of any type, as long as they support wireless communication. This communication can be through any kind of protocol, although usually Wi-Fi or Bluetooth connections are the weapons of choice used when an actual implementation needs to be done.



An example of an ad-hoc network involving PDA's, laptops and mobile phones.

Although not exactly equal, we could compare ad-hoc networks to P2P networks, as they both are systems in which they're participants send and receive information in a non-centralized way.

One of the many significant difference between a MANET and traditional mobile networks (like cell-phone providers networks: 3G, GPRS, etc) is that they provide a significantly bigger deal of complexity to they're deployment. For instance, they're topology is constantly changing, as a user may go in and out of a MANET's coverage area several times in a significantly small amount of time.

The lack of infrastructures that provide connectivity is then, ad-hoc weakest and strongest feature. While it makes it a particularly difficult asset for commercialization, the versatility gained can be, in determined situations, critical.

Delay Tolerant Networks

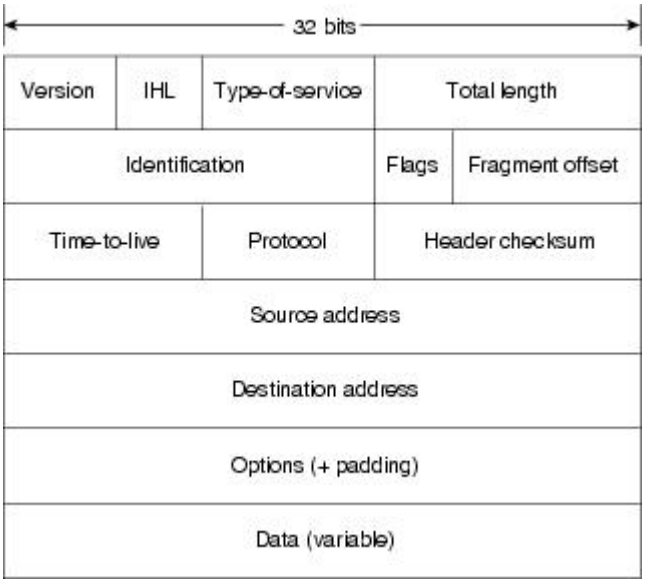
Today's internet

When the TCP/IP protocol was designed, it was bearing in mind that connections would primarily rely on wired links, including the telephone network. Although nowadays we do have some reliable wireless technologies that are proving to be trustworthy, the general assumption is that links are continuously connected in end-to-end, low-delay paths between sources and destinations. They have low error rates and relatively symmetric bidirectional data rates.

Packet switching

Packet switching is the basis of the communication over the Internet. A packet is a piece of data that assembles with many other packets to form a block of user data that has to be delivered from one user to another. For instance, when we download an HTML webpage, we don't download it all as a bundle, but we download all the different packets instead.

Packets travel independently from each other, and may take different router from they're source to their destiny. The source and the destiny of a packet are commonly called nodes.



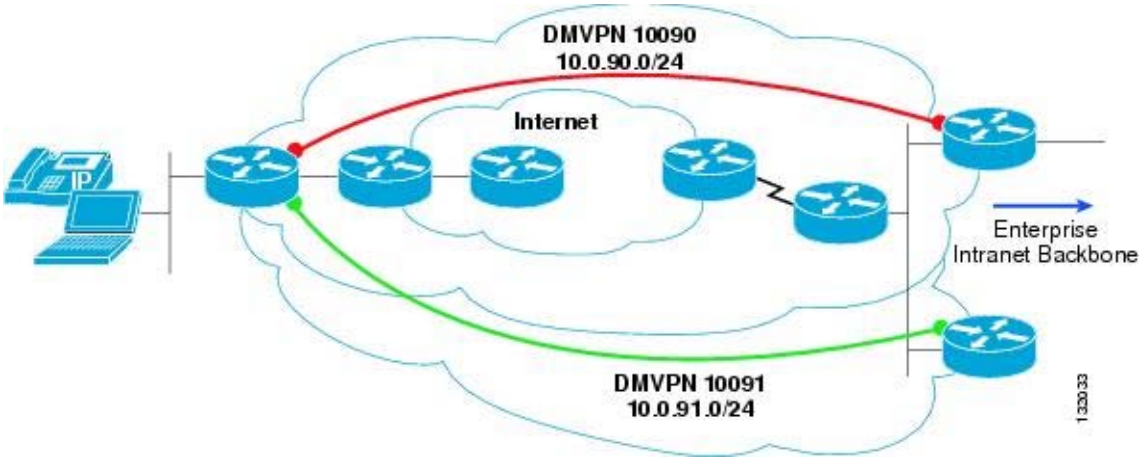
A TCP/IP Packet

Packets follow a route of links throughout the network. Whenever a link fails or it is disconnected, packets should find another route by which they will arrive to their destiny. They also have both the header and the user data (payload), and it is the header that describes how the packet should be switched from one router to another. Although it is probably that packets arrive out of order, they are correctly assembled at their destiny.

The usability of this paradigm is based on some assumptions:

Continuous, Bidirectional End-to-End Path: A continuously available bidirectional connection between source and destination to support end-to-end interaction.

Short Round-Trips: Small and relatively consistent network delay in sending data packets and receiving the corresponding acknowledgement packets.
Symmetric Data Rates: Relatively consistent data rates in both directions between source and destination.
Low Error Rates: Relatively little loss or corruption of data on each link.



Packet routing

Breaking with the assumptions

There is an underlying issue to the internet paradigm, and it is that many networks may not be able to conform to the Internet's general assumptions. This may be due to several reasons, and may incent us to consider whether they would work at all if we didn't bend the rules for them. These networks are characterized by:

Intermittent Connectivity: If there is no end-to-end path between source and destination (also called *network partitioning) e*nd-to-end communication using the TCP/IP protocols does not work.

*Long or Variable Delay: L*ong propagation delays between nodes and variable waits and delays at nodes contribute to end-to-end path delays that can mess up with Internet protocols and applications that rely on quick responses.

Asymmetric Data Rates: The Internet supports moderate asymmetries of bidirectional data rates, but if asymmetries are large, they defeat conversational protocols.

High Error Rates: Bit errors on links require correction (which requires more bits and more processing) or retransmission of the entire packet (which results in more network traffic). For a given link-error rate, fewer retransmissions are needed for hop-by-hop than for end-to-end retransmission (linear increase vs. exponential increase, per hop).

What is a DTN

A delay-tolerant network (henceforth DTN) is a network of regional networks. It is by definition a layer that works on top of regional networks, even including the internet. A DTN supports interoperability of different regional networks by broadening the specter by which Internet connections are limited.

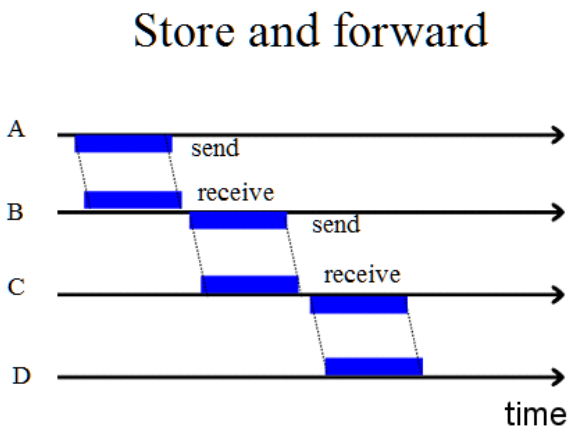
A DTN supports long delays between and within networks, and can be able to translate between protocols if several regional networks have differences in their behaviors. Basically, a DTN will try to address the technical issues in heterogeneous networks that may lack continuous network connectivity. Examples of such networks are those operating in mobile or extreme terrestrial environments, or planned networks in space.

Store-and-forward

Intermittent connectivity, variable or long delays, error rates and asymmetric data rates can be overcome in a DTN by using *store-and-forward* message switching. This method is really not new, and has been used to achieve message transmission from ancient times, as postal systems used to implement it. The idea behind this method is to continuously store and send the message that we intend to reach its destiny to different paths that can eventually reach it. Actually some e-mail systems or voicemails use this method, although in a slightly different way, as they use a central server to store the messages.

The difference between this method and the usual Internet routing is that store-and-forward provides storage for the message for a much wider space of time (in can even be indefinitely). The storage needs to survive through some time because:

- There may not be a link to the destiny for a long time.
- We are not sure if the message has been transmitted or there have been some errors along the way.
- Data transmit rates can be very variable.



Intermittent Connectivity

Connectivity intermittence can be caused by several motives. The most common are constant mobility that can lead to a distance too high as to assure a connection and power issues. Another reason can be signal blocking; for instance, in interplanetary connections, links can be obstructed by the appearance of big bodies in between the nodes.

If we were to use the Internet's routing protocols, intermittent connectivity would lead to loss of data, as packets that cannot be immediately forwarded are dropped. If packet dropping is severe enough, there will even be a loss of connection, as TCP works that way, and that will result in applications not fulfilling their purpose or even failing.

DTNs use the store-and-forward technique to overlap this issue, as they can isolate delay.

Opportunistic routing

There are many types of DTN protocols. When we can somehow schedule the availability of a node in the network we are talking about scheduled or predictable contacts. . For example, in interplanetary communications, many times a planet or moon is the cause of contact disruption, and large distance is the cause of communication delay. However, due to the laws of physics, it is possible to predict the future in terms of the times contacts will be available, and how long they will last.

In other situations, such as emergency responders, we may not know how the network will evolve, as nodes behave in a random way. Contacts that participate in these connections are called intermittent or opportunistic.

For instance, moving people, vehicles, aircraft, or satellites may make contact and exchange information when they happen to be within line-of-sight and close enough to communicate. Opportunistic contacts happen all the time: If we want to talk to a friend and we happen to cross with him across the hall, it is an example of an opportunistic contact. We can program devices to behave in certain ways when crossing with other devices. For instance, a cell phone can be programmed to send or receive information when crossing certain zones in the city, and exchange this information with other cell phones.

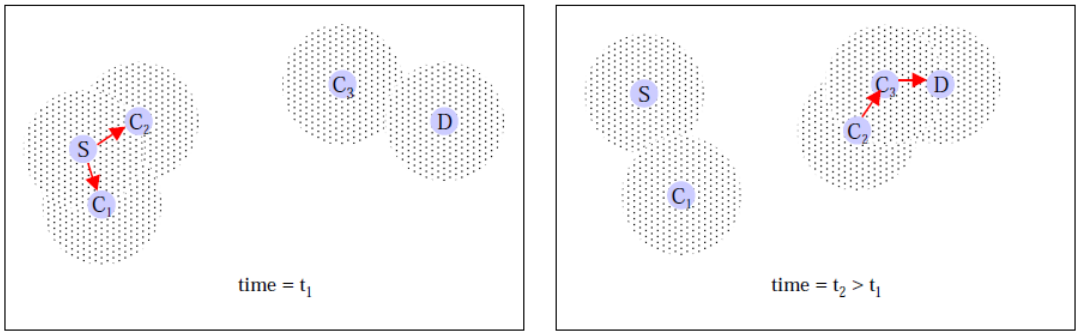
Epidemic Routing

Epidemic routing attempts to maximize the data arrival in a network that most probably never will be fully connected. It is part of the opportunistic routing, and attemps to make minimal assumptions about the connectivity of the underlying ad-hoc:

1. The sender node is never in range of any base station.
2. The sender does not know where the receiver is currently located or the best route to follow.
3. The receiver may also be a roaming wireless host
4. Pairs of hosts (not necessarily the sender and the receiver) periodically and randomly come into communication range of one another through node mobility.

The idea is, then, to distribute the data between partially connected networks, by sending this data through hosts that will act as carriers of it. We are relying, then, in carriers coming into contact with other portions of the network through node mobility.

The message sent will, therefore, be sent to a number of islands of nodes, as much as possible, having a relatively high probability of eventually reaching its final destination.



(a) (b)

In the example above we can see how the sender, S, attempts to send a message to D, the receiver. S relays the message into C_1 and C_2, having C_2 eventually crossing paths with D and therefore achieving a correct delivery.

Application Basics

Our application started up because of some experiments that were being developed in the UPC. In this experiments in which our research group participated there was a deployment of an application based in Miraveo's technology. Miraveo provided quite a variety of scenarios to test the application: a chat, a ringer and file transferring, and proved to be a reliable library that could offer something that the community could be interested on.

The only issue was that, whilst Miraveo's application worked quite fine, we couldn't really tackle any serious data metrics from the experiments logs. We couldn't because, as we knew later, there was no real "Delay tolerance" in Miraveo's application, in the sense that users were identified by an IP that was not persistent when they reconnected to the network later during the day.

Therefore, we started playing with the idea of a basic application that would be used for testing purposes, and would provide a way to actually extract some data from the experiments we would realize.

Thus, it was decided that to do so, we would create an application that would allow playing with Miraveo's library in a way that would make it easy to modify it to test different routing algorithms. It would initially use an epidemic algorithm, but would grow bigger in a future when some test runs would have been made.

Being limited by the library's supporting technologies, the application would have to be developed using Qt under Symbian.

Basic application concepts

Push technique

Miraveo's push method is basically a method in which a user A can send Delay Tolerant Data Objects (from now on DTDO's) to another user B if there's visibility between them, by constantly trying to establish a link. Consequently, this method will only work if both users A and B appear to be in the same network at the same point of time, or else it will not be possible to succeed by sending a message using this technique. The delay tolerance in these could be simulated by constantly trying to re-send the message during a determined amount of time, but lacks sense when we are also given the opportunity of using the pull technique.

Push technique will be used in messages that need immediate confirmation or presence of both users in the network. In our application, we use this method to send friend and group notifications, so that we are sure that all users in the network are aware of they're environment. This is important because it is the only way that users will be able to make new friends or be able to discover new groups that they can be part of.

In the application use cases we will see several examples of this technique.

Pull technique

The difference between push and pull technique is basically that push techniques involves the sender of the message explicitly giving the DTDO to the message's recipient, while pull technique will require of the recipient node to actively search whenever there is a message for him, and ask for it if he finds so.

While push technique may seem more natural in the traditional messaging paradigm, using the pull technique will give us farther more versatility and an edge that push technique couldn't possibly do (or could, but with much more work). By using pull technique we can achieve real disconnected delay tolerance in our message sending. Therefore, if a node A is trying to send a message to send a message to node B, he doesn't has to actually see him to do so, or even be in the same network or the same time frame.

Pull technique requires all nodes in the network to 'publish' some DTDOs that they want to share. When finding other nodes, they will ask each other what are the DTDOs that they are sharing, and decide which of them should they request and re-share or keep them if they are destined to themselves. This way a message from A to B could arrive even if node A haven't seen node B in a long while but they, for instance, have a friend "node C" that they both see.

The handling and relaying of these messages is obviously non-trivial, and requires quite a preparation that we will comment in its own section.

Tag identification

Miraveo's library allows us to create DTOs that we can publish over the network. The creator of these DTOs is in the form:

```
MVNetworkConnector::createDTDO(MVLocalDTDO*& dtdo, uint32_t tag,
const QByteArray& data, int32_t ttl);
```

These DTOs are identified by their tag and their origin IP, and must be unique in the network. We can see that the issue here is to identify which messages are directed to us using the 32 bits tag. We first thought of an approximation based on using masks inside the tag so that we could filter the tags by mask and be able to distribute them correctly.

The issue with this method is that each tag has to (obviously) be unique in the network. This would mean that for each friend or group we should use some of the bits destined for the tag to identify the message and some other to mask the origin (as to identify it). Moreover, the tag identifier (the part that's not the mask of the tag) should be random, as it is not possible to ensure the coherence of a sequential method.

All this sums up in a very high possibility of collision between our DTOs, forcing us to search for a viable alternative that will let us transmit all the information that we intend to give to other users in the network as to be able to make informed decisions about which DTOs should they get or leave.

Second approach: METADTO

A possible solution to all this problem is using a "know DTO" that contains data about all other DTOs that we are sharing (or we have and could share). Therefore, what we would do is have every user continuously querying themselves about what DTOs do they have, and which moves do they make (getting DTOs, publishing them, etc) and have every one of this DTO have a random tag, except the 'metaDTO' that will tell us which information each of this other DTOs has.

For instance, let's say that a user A meets another user B (again). User A would not ask for the list of all B's tags, although it may be natural, but instead ask for a DTO with a known tag that has information about all other DTOs that B has, thus allowing A to have much more information on every DTO that B has available for sharing. Once A has asked for this DTO, he can analyze it and we will be able to make well-informed decisions to which DTOs should A ask B for. Therefore, we don't have to decide 'a priori' on a request policy and hope for the best, but we can actually try several of them and see which one works better for us in different situations.

This method will force every user to be sure that they maintain a coherent 'metaDTO'. Consequently, every time a user publishes or shares a DTO they must be sure that they refresh the 'metaDTO' in order to allow other users to have a reliable vision of what's what in our network.

Final approach: Key uniqueness

The main issue we encountered when using the metaDTDO approach is that it was programmatically uncomfortable to keep the metaDTDO updated, especially because of the

messages time to lives, which had to be periodically checked for us not to have as available a DTDO that shouldn’t have to be anymore in the network.

We came up with a solution that provided us with the best of simplicity and usability: We created for each message a random key that would identify it along the network. Although there can be an issue of message colliding, having more than one having the same id is highly unlikely and, in the end, an acceptable risk. We will

Qt programming

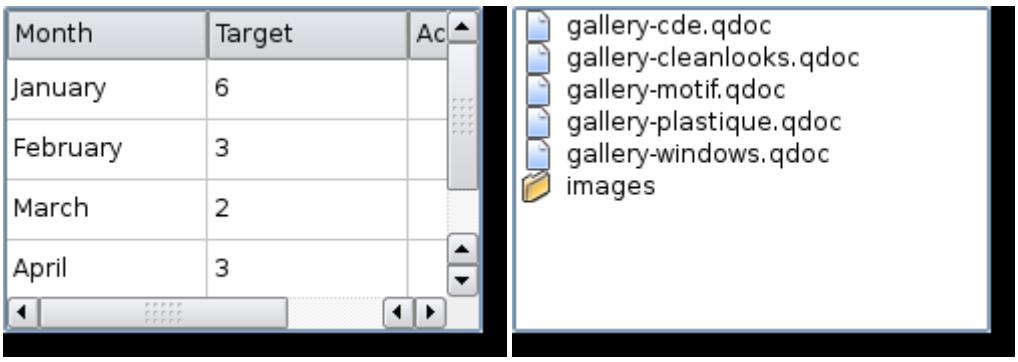
Qt (pronounced as cute) is a cross-platform application framework that is widely used to develop application software with GUI. It uses standard C++ but extends its capabilities by using a code generator (called the *Meta Object Compiler*, or *moc*) together with several macros to enrich the language. It runs on all major platforms and has extensive internationalization support. Non-GUI features include SQL database access, XML parsing (that we will use extensively), SQL management, network support, and a unified cross-platform API for file handling.

GUIs with QT

Widgets are the visual components that we will use to create the user interfaces. They are basically all visual elements we can refer to: Buttons, menus, scrollbars, message boxes, application windows, tables, and many others are all examples of widgets that we can use in our application.

Layout managers organize widgets, being in charge of making the child widgets stay into the parent's widget's area. They can perform automatic resizing and positioning of the elements that are included inside them.

All components are connected to other elements by using a simple signal-slot paradigm that is available for all of them (even our custom ones).



Examples of widgets

Layouts

When designing forms using Qt, you do not place the widgets at static locations. Instead, you place them in layouts. This makes the user interfaces stretchable and allows them to adapt to the current screen and contents.

It is usually a better idea to use dynamic instead of static layouts, as screens may have different resolutions, aspect ratios, etc. Different systems use different fonts and may have different styles.

A situation that would benefit from using dynamic layouts is, for instance, when we have a table that we don't know the content of, or when we want to translate some text (text in Finnish may not be the same width as text in Spanish).

Horizontal layout

Vertical Layout

Grid Layout

Signals and Slots

Signals and slots are the event manager in which Qt is based on. They allows you to connect events to slots so that you can answer to button clicks, checkboxes being checked, text being edited – but also to values being changed, timers timing out, etc.

Every time we want out Qt application to perform an action we usually have to connect signals to slots. We can connect signals and slots:

By name. This means naming the slot *on*_widget_signal. This connection is made when you call *setupUi* and lets you connect simple triggering events to a slot. Although it is quick and easy, it is not really recommended due to legibility and usability reasons. Also, you can make this type of connections by right clicking in the Designer view of QtCreator.

Using the connect method. By specifying a signal source object, the signal name, a destination object and a slot name you can connect any signal to any slot.

Then, there is the last way:

Connecting using Designer. You can actually draw your connections by using Qt's GUI Designer tool.

Experience using Qt

Qt isn't the most easy-to-use platform we can come across. Although it is based in C++, its extensions, and the fact that most built-in Qt functionalities need them to work appropriately,

make the learning curve far steeper than it should be being a framework based in a language that broadly known.

We could probably state that it isn't really the fact that QT is difficult per se, but a bit too laborious. While signals and slots work quite fie and allow us to fine-tune our application, other approaches (as Adobe's with Flex or Google's with GWT) may result in a simpler use of the connecting system.

On the other hand, Qt is quite easily readable, and can be taken over by other programmers if necessary, a focal point here in this project. It also offers some nice features, like the Xml parsing, that we've actively used during the project, and have proven to be very useful.

Documentation

One thing I must highly remark about using Qt is the fact that its documentation is really well-structured and relatively easy to follow. It is a simple task, finding examples of what we want to do with our application.

An improvement on this documentation was done since Nokia bought Qt, as they need a clear documentation to fight their competitor's more advanced frameworks.

Looks

Maybe one of the greatest handicaps of Qt is the fact that they use every system's native look and feel and don't add much to it. Compared with other frameworks, an application based in Qt looks, if we don't put a lot of work in aesthetics, will look a bit poor. While this isn't determinant to our application, as we are interested in the functionality results, it may be a stopper for most developers using this framework, as commercial applications usually sell not only because of their functionality, but also (and maybe even more that functionality itself) because of their eye-candy.

Developing for Symbian

Developing for Symbian using Qt is a fairly easy task. As Qt is a multi-platform framework, we don't have make any important or significant change to it in order to work. The only things we must bear in mind are screen sizes, application Ids and softkeys.

- Screen sizes are usually different from desktop applications, we must bear in mind not only the legibility but also the aspect ratio when developing for symbian.
- Symbian uses a number of globally unique 32 bit numbers (known as unique identifiers or UIDs) to identify file types and executables. Qt application developers need generally only concern themselves with the UID3 value, which uniquely identifies a specific executable.
- Softkeys can be used as part of the application (they usually can't be taken away from the screen actually). We can use them to provide the basic navigation in our application.

Options and Exit are the softkeys that we will be using to interact with the application menus.

State of the art in mobility mobile social networks

In this section we will go briefly over the papers that helped us arrive to the conclusion of what would be valuable in our application. They are testbeds that have been done in several universities, all of them using their own system and metrics.

"Access and mobility of wireless PDA users"

This paper, developed at the UCSD (University of California, San Diego), was written by Marvin McNett and Geoffrey M. Voelker. In it, they described a study in which they analyzed the mobility patterns of handheld PDAs using an 11 week trace of wireless network actrivity.

They had three goals:

- Characterize the high-level mobility and access patterns of handheld PDA users and compare them to previous studies that were originally focused on laptop user.
- Develop two wireless topology models for use in the wireless mobility studies: an evolutionary topology model based on user proximity and a campus waypoint model that served as a trace-based complement to the random waypoint model.
- Finally, they used the topology models to study ad-hoc routing algorithms on the network topologies.

They logged the visibility of the Access Points, and assumed that two nodes were in contact if they were at the same time in coverage of the same AP. They recorded the MAC addresses, the signal strength, the associations and the power state.

The tests went on during 77 days, with 273 devices being monitored, and they achieved some interesting metrics:

- The granularity of the refresh was 20s
- The number of internal contacts was 195364
- The average of contacts/pair/day was 0.034

This testing was characteristic due to the fact that they only logged the visibility of the APs, so it needed an infrastructure to work.

Some of the results of the experiments were:

- Locations of the devices could be estimated by using a triangulation and empirical correlations based on the APs signal strengths.
- They had to use a simple heuristic time when detecting and calculating session duration, as weak signals made the connection fluctuate and would appear as a big quantity of weak contacts. They used a time between 1.5 and 4 minutes to determine that the session was new.
- A noticeable decrease of the use in the system was noticed, even more that the one they had originally expected. This may have been because of:
    - Battery draining, causing a system restore and therefore a deletion of the software they used.

- Declining interest in the PDA
- Declined interests in the network, so they turned it off (although they did use the PDA).

"User mobility for opportunistic ad-hoc networking"

This paper was written at the University of Toronto by J. Su, A. Chin, A. Popivanova, A. Goel, and E. de Lara. In it, they maintained that they could not only create an ad-hoc network, but also have an intelligent routing that could be power efficient, if they could presume that packets were latency-insensitive.

They ran two studies collecting user mobility data by giving instrumental PDA devices to groups of students. They had to carry these devices for several week. They were attempting to get empirical data that proved that it is possible to make intelligent routing decisions based on only pair-wise contact, without previous knowledge of the mobility model or location information.

In the experiments, they didn't worry about transfer data, connection quality, and bandwidth or user location. The device they provided the students with carried useful functionality to encourage them to carry it, and the software was executed invisibly in the background with minimal impact on the usability.

In their experiments, they selected most active pairs of nodes to evaluate the end-to-end communications, using an epidemic dissemination of the packets, and having into account just the first packet arriving to the destination node. Devices used in the experiments were NTP (Network Time Protocol) synchronized so that radio usage could be minimized and the odds of successful communication increased.

They based their metrics on:

- Multi-hop packet delivery Latency
- Number of multi-hop meetings between node pairs.
- CDF of multi-hop packet arrivals for the selected pairs of nodes.
- Hop count proportion

The conclusions they arrived to included:

- Work day may not be enough to take enough data if not having a good user-base.
- The problems they had were basically because of battery issues. An effective testing can only be done if devices survive battery drainage or can be constantly recharged.

Some interesting points from their metrics were:

- They used Bluetooth connections
- The granularity of the refresh was 16s
- The number of internal contacts was 2802
- The average of contacts/pair/day was 0.35

"Reality mining: Sensing complex social systems. Journal of Personal and Ubiquitous Computing"

This experiment, driven by N. Eagle and A. Pentland for MIT, proved to be interesting in many ways. Firstly, it was due to the fact that it attempted to study human behavior and patterns, as it was a study for social sciences. Secondly, it was a 9-months test, which provides a great amount of data to benchmark.

The software designed to run this experiment was running as background, and could be installed in any java-capable phone. They used a base of 100 Nokia 6600 phones that were used over the campus.

In these experiments they had a compromise between refresh rate and battery life: they would scan every 5 minutes to try and prologue the battery life of the devices.

The most interesting conclusions we take out of this paper are:

- Data can be corrupted if stored in a flash card with a limited number of read-write cycles.
- An obvious correlation can associate friends/family by studying the logs.
- Most of the time data anomalies could be associated with human related errors (forgot phone, vacations, battery, etc.)

"Impact of Human Mobility on the Design of Opportunistic Forwarding Algorithms"

In this paper, produced for the University of Cambridge by A. Chaintreau, P. Hui, J. Crowcroft, C. Diot, R. Gass, and J. Scott, they observed that the distribution of the time gap separating two contacts of the same pair of devices exhibits a heavy tail (power law). To do so, they based their study in 6 studies (data sets).

The most interesting part about the paper was the statistics they had about the studies:

User Population	Intel	Cambridge	Infocom	Toronto	UCSD	Dartmouth
Device	iMote	iMote	iMote	PDA	PDA	Laptop/PDA
Network type	Bluetooth	Bluetooth	Bluetooth	Bluetooth	WiFi	WiFi
Duration (days)	3	5	3	16	77	114
Granularity (seconds)	120	120	120	120	120	300
Devices participating	8	12	41	23	273	6648
Number of internal contacts	1,091	4,229	22,459	2,802	195,364	4,058,284
Average # Contacts/pair/day	6.5	6.4	4.6	0.35	0.034	0.00080
Recorded external devices	92	159	197	N/A	N/A	N/A
Number of external contacts	1,173	2,507	5,791	N/A	N/A	N/A

.

Application insight

In this section we are going to give a detailed explanation on the application's implementation. This will be given for every expected use case.

Objects

We mainly have two kinds of different objects used in our application:

- Qt objects, widgets that are used for the logic and for visual representation
- C++ objects, that we will use for data persistence and are the objects that will form part of the

Persistent objects

Our application has, mainly, four kinds of objects that need to be persistent:

- Users: they will represent the instances of each user connected to the network
- Friends: Very similar to users, they will represent the users that we have befriended all throughout the network
- Messages: The actual messages sent by the users through the network.
- Groups: Associations that many users can share to be able to receive the same message.

These objects are, in our system, the objects that will travel through the network. They will have to have the ability to be parsed to an object and sent so that they are correctly received by the users. They will also have to be easily understandable by the application so that they can be 'treatable'.

Widgets

Widgets are the main screens of our application. They apply the logic that will allow us to successfully fulfill all the use cases, and provide the visual representation that the users will need in order to be able to interact with it. We have several widgets, each of them with their own functionality. The widgets used are:

- Connect Widget: The widget that will appear to the user when they boot up the application. Will provide a basic interface that will allow the user to identify him, as well as to register. It will interact with the main widget to create the connection to the network.
- Widget Controller: This is the main widget that will provide most of the logic of the application. Not only will it control principal screen changes, but also will control the main network functions. Here is where we decide the routing policy that we want to follow.
- Main Selector Widget: The main selector will be the main screen we see after connecting. It will allow us to select between user and group management, and will let us consult the inbox.
- User Management Widget: User management is a transition screen that will let us choose between seeing our friends or the users connected to the network.

    - Friend list Widget: This widget will allow us to consult our friends, see they're details, or send those messages.
    - Network Users Widget: Similar to the friend list, this widget will allow us to consult the users of the network, send those messages, and befriend them.
- Inbox Widget: The inbox widget will allow us to consult all the messages that have been sent to us or to one of the groups that we are part of.
    - Message details widget: Message details widget will show us some details on the messages that we have received, and will allow us to add the sender to friends, answer the message, or go to the inbox page.
- Main Group Widget: Transition widget that will allow us to go between some options:
    - Create Group Widget: Widget that will allow us to create a group and publish it in the network.
    - Current Groups Widget: Widget that allows us to consult the widgets that we are subscribed at.
    - Available Groups: Widget that will allow us to consult the widgets that are available throughout the network.

Issues with persistent objects

The first implementation of every object was very similar to the one we ended up with, but had a major difference: it was parsed to a byte data array and sent as-is via the network. This worked quite fine when doing some simple tests but it ended up creating a big issue in the system: we were attempting to successfully achieve creating an implementation that would be used by the research group in a reasonably easy way, so the transition shouldn't be really painful.

Sending the objects this way, and storing them like this, proved to be a complicated task. It was complicated not because it was especially difficult in a programmatic way, but because it created a high added complexity when trying to associate messages to users, users to friends and groups, etc.

The first implementation of the system delegated all this work into the main widget, that would parse all the files and look for attributes that would define every object's correct owner and relations, but it proved to be unreasonably complex. The system worked fine, but the code was a big and complex piece, and could represent an issue when adding or deleting attributes, or creating an added class to be inserted in the system

A second idea revealed to be much more elegant in terms of coding: we created files for the objects, in a way that filenames could be deduced by every user's key, and all related filenames for that user were also calculated in a way that would allow us to easily access every user's information in a non-painful way. Also this seemed to work fine, there was quite a mess in the file system. In addition, the system wasn't really elegant, and I couldn't assure that there wasn't really going to be an issue with it in some way.

Finally, I toyed with the idea of using some language that would be easily searchable so that I could embed more info in a single file and simply the search and injection of data. To do so, I looked into QT's supported data models and saw that they natively supported working with Document Object Models (also known as DOM) in quite an efficient way.

Document object model

The document object model is an API that's used for both HTML and XML documents (providing they're well-formed). By defining the logical structure of the document, the API knows how to access and manipulate it, so that it can be a lot easier for us when dealing with a great amount of data.

In our case, XMLs are used to represent many kinds of different information that are stored in the filesystem, but once accesed, can be easily managed by the DOM API, so that we can greatly increase the ease of use of the system.

Using the Document Object Model, programmers can create and build documents, navigate their structure, and look for, modify, add or delete elements or content.

An example of how DOM sees elements would be that of the parsing of an XHTML file:

```
<table>
  <tbody>
    <tr>
      <td>Shady Grove</td>
      <td>Aeolian</td>
    </tr>
    <tr>
      <td>Over the River, Charlie</td>
      <td>Dorian</td>
    </tr>
  </tbody>
</table>
```

A graphical representation of how DOM sees this object would be:

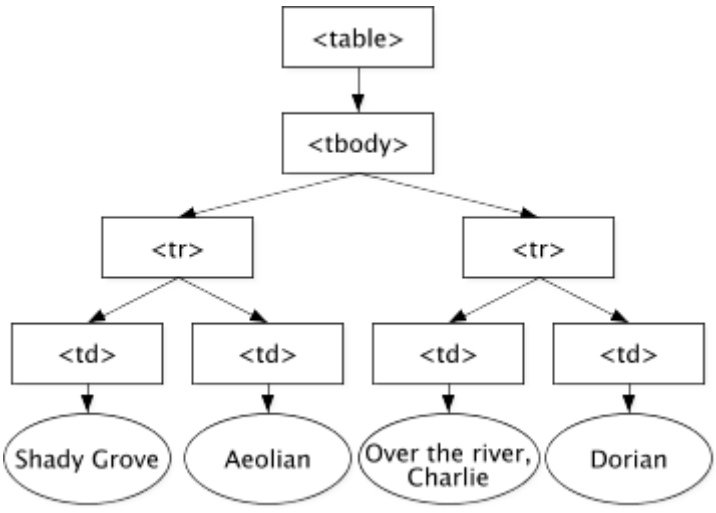


Figure extracted from http://www.w3.org/TR/DOM-Level-3-Core/introduction.html

We can see how DOM orders our data hierarchically, and orders it by tags. We will precisely be using this tags to create a human-readable and programmatically simple object model that will allow us to work with ease.

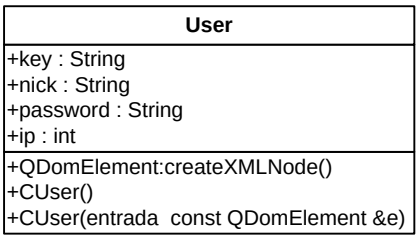
Persistent Objects

Users

Users are our basic connection and visible objects found through the network. They represent the instance that will be exchanged by every client that is being connected to the network, and will be 'uniquely' identified.

A user is represented by:

- A *key*, that will uniquely identify him through the network. This key is a randomly generated 64-bit integer (parsed into a string for commodity purposes) that will allow us to route messages successfully to their final destination.
- A *nick*, that will be the humanly understandable identifier for the object itself. Other user will see him by this name, although they will also be able to identify him via his key if they want to.
- A *password*, that will assure us that once a user is registered, there's not another user using the same device that can, deliberately or not, use the same nick and therefore receive his messages.
- An *ip*, that will be given to us when a connection is realized. It is included for convenience.



A user has several operations he is in charge of:

- A basic constructor that will create a User object.
- A constructor that will create a User having as a parameter a DOM element. This is implemented in every object that has to be sent through the network or saved in a file, for convenience reasons.
- An operation that will allow us to create a DomElement from a User in a direct way. This, just as the above operation, is also implemented in all the objects that are sent throughout the network, and proves to be quite convenient.

Every registered user in the network will, internally, be saved as a User in a device.

Friends

Friends are an extension of the user class, but also have an *ownerKey* attribute. This attribute will internally allow the application to discriminate which user can see this member as a friend and who can't.

Friend
+key : String +nick : String +password : String +ip : int +ownerKey : String
+QDomElement:createXMLNode() +Friend() +Friend(entrada const QDomElement &e)

All other attributes have the same values as a Friend class. We keep password value for commodity but won't really be giving any use to it.

Just as in the User class, we have:

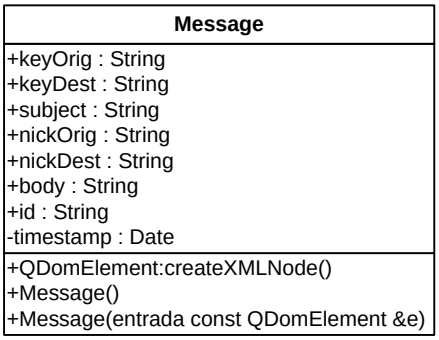
- A basic constructor that will create a Friend object.
- A constructor that will create a Friend having as a parameter a DOM element. This is implemented in every object that has to be sent through the network or saved in a file, for convenience reasons.
- An operation that will allow us to create a DomElement from a Friend in a direct way. This, just as the above operation, is also implemented in all the objects that are sent throughout the network, and proves to be quite convenient.

Friends are the representation of the social relations that every user will have in the network. Befriending someone allows us to send him messages even when he is not in the network at that precise moment (although, as we already know, we cannot ensure that he will ever receive the sent message).

Messages

Messages are the main structure that will be sent throughout the network. They represent all the data that users will intend to send to one another, and are the main pillar for the usage of the application. A message is represented by:

- An origin key, *keyOrig*, which represents the unique identifier that belongs to the sender of the message.
- A destiny key, *keyDest*, which uniquely identifies the intended receiver of the message sent.
- A *subject,* which can be used to add subjects to the messages in order to provide a better organization.
- A destiny nickname, *nickDest,* that, if known, will represent the intended receiver's nickname, used for aesthetic reasons.
- An origin nickname, *nickOrig,* used to identify the owner of the message. It is especially useful if the receiver does not know the sender's key.
- A *body* field, which will contain the message itself.
- An *id,* or unique identifier, that will intend to uniquely represent the message. This will prevent us from receiving the same message more than once.
- A *timestamp* field, that will allow us to know when was the message sent.



Just as in the other object classes, Messages have implemented:

- A basic constructor that will create a Message object.
- A constructor that will create a Message having as a parameter a DOM element. This is implemented in every object that has to be sent through the network or saved in a file, for convenience reasons.
- An operation that will allow us to create a DomElement from a Message in a direct way. This, just as the above operation, is also implemented in all the objects that are sent throughout the network, and proves to be quite convenient.

Groups

Groups are the final basic object that we will be using in our system. Groups are a representation of associations of users. They are basically threads that users can subscribe to, in order to be able to receive messages that other users also receive, and be able to answer them. This way, we can create communities that share interests.

Groups are represented by:

- A *key*, which will uniquely identify them throughout the network. This key is very similar to those Users and Friends use.
- An *id*, that will basically be the group's name.
- A *topic*, that represents the theme that the group will be meant to talk about. For instance, a group with a "Volleyball fans" topic will lead all volleyball fans to want to be part of it.
- An *ownerKey*, that represents, in a similar fashion as in Friends, the *key* of the User that is subscribed to this messaging group.

Group
+key : String +id : String +topic : String +ownerKey : String
+QDomElement:createXMLNode() +Group() +Group(entrada const QDomElement &e)

Last but not least, Groups implement:

- A basic constructor that will create a Group object.
- A constructor that will create a Group having as a parameter a DOM element. This is implemented in every object that has to be sent through the network or saved in a file, for convenience reasons.
- An operation that will allow us to create a DomElement from a Group in a direct way. This, just as the above operation, is also implemented in all the objects that are sent throughout the network, and proves to be quite convenient.

Main Use Cases

The application has a broad amount of use cases. We will here comment only those more interesting and relevant to this documentation. The intention is to comment on all those use cases that will be typically used by a standard usage of the application.

User identification

In the identification use case the user we have a scenario in which a user will want to identify himself. To do so, he will use the *ConnectWidget*, and state a username and password that will define him as a user.

1. The user opens the application and states a username and a password
2. The widget opens the file in which authentications are stored and creates a DomDocument that represents this file.
3. The widget searches, in this file, for the credentials that the user had used, and matches the username to the password.
4. The widget send a connect() request to the *WidgetController* and changes the screen to the main screen.
5. The user is connected and able to start using the application.

Extensions

The user and password don't match:

1. The system presents a screen that states that there's a user and password mismatch
2. The user introduces new credentials
3. The widget starts with main action's 2nd step.

The user doesn't exist:

1. The widget creates a new DomElement that represents the user that will be created.
2. The widget appends this information to the users credentials documents so that they will be remembered in the future.
3. The widget sends a connect() request to the *WidgetController* and changes the screen to the main screen.
4. The user is connected and able to start using the application.

Sending a Message

Sending a message is one of the basic scenarios that a user will encounter, and is fairly simple. A message can be sent only from the *MessageSend* widget, which can be accessed by various means:

- From the *NetworkUsers* widget
- From the *FriendsList* widget
- From the *CurrentGroups* widget
- From the *MessageDetails* widget

All this widgets allow us to select a receiver to whom send a message. This receiver can either be a single user or a Group (either one we've subscribed to or one that we created ourselves). Depending on whether the receiver is connected or not, the message sending will be done using the pull or the push method.

To clarify it, it would follow these steps.

1. The user ends up in the *MessageSend* widget screen following one of the paths described above.
2. The system automatically detects where the user came from and fills in the "To:" label with the nickname of the user or group that the user intends to send the message to.
3. The user introduces some text into the body of the message and presses the "Send" button.
4. *MessageSend* widget creates a message object with the details of the message that is going to be sent and sends a request to the *WidgetController* to delegate the message transmission.
5. The user receives a notification about the message sending, indicating whenever it has been sent directly to the user or, on the contrary, if it has been published.

Checking a message

Messages can be consulted at the *InboxWidget*. They will represent all the messages in the network that the user is allowed to consult. Given a user that just connected to the network, the use case would be:

1. The user clicks on the “inbox” button located on the *SelectorWidget*
2. The system populates the list of messages for the user and presents the *InboxWidget* screen populated
3. The user scrolls through the list, where he can fins his personal and group messages, and clicks on one of them
4. The system returns a *MessageDetails* widget populated with the contents of the message clicked.

Creating a group

To create a group, we must go to the *CreateGroup* widget, in which we will we able to create an instance of a group that will be shared over the network. Given a user that just connected to the network, the use case would be:

1. The user clicks on the “Groups” button.
2. The system changes the screen to the *MainGroupWidget*, where the user can select the different group options
3. The user selects to create a group by clicking on the “Create” button
4. The system changes the screen to *CreateGroupWidget*, where users can enter the details
5. The user enters the data that he wants the group to contain. He must enter an id and a subject.
6. The system creates a random key for the group and creates the Group object. It stores the data in the file system, publishes the group, and subscribes the user to it.
7. The user is now part of the group he created, and publishes it with other users.

Checking current groups

The user can consult the groups he is subscribed to at anytime in the *CurrentGroupsWidget* screen. To do so, the process must follow some steps:

1. The user clicks on the “Groups” button.
2. The system changes the screen to the *MainGroupWidget*, where the user can select the different group options
3. The user selects to consult his groups by clicking on the “Current” button
4. The system checks the groups that the user is member of and populates the *CurrentGroupsWidget* with a list of details on them.
5. The user can see the groups he is member of

Checking available groups

The user can also consult the groups that other users share throughout the network. To do so, he must follow this use case:

1. The user clicks on the “Groups” button.
2. The system changes the screen to the *MainGroupWidget*, where the user can select the different group options
3. The user selects to consult the networks published groups by clicking on the “Available” button
4. The system populates the list of current published groups in the *AvailableGroupsWidget* and presents the widget to the user
5. The user can now check the available groups in the network.

Network handling

Developing using Miraveo's library

Miraveo's network is an example of spontaneous proximity wireless network. This network allows us to discover other users that are also part of it, and will ease the communication between nodes. When developing using Miraveo's network, we must be very aware of the network itself, and change the paradigm that we are following for one that applies for this type of network, that can be really different to those that we are more used to.

Basic Miraveo's functionalities include methods to connect to the network and be aware of the surroundings at all times. That means that we can handle in a relatively easy way user discovery and mapping the network. Whenever a user connects to the network, he signals all the other neighbors in the proximity that he has joined the network, and is allowed to discover users that are already connected. In a similar fashion, users are signals when someone leaves the network.

Probably the most interesting feature that characterises Miraveo's library is the flexibility we are given to implement different communication paradigms. The data exchange mechanisms that are given to us are those that we need to implement a network that can be tolerant to delays and disconnections. Furthermore, we will be able to send data to nodes that aren't present in the network at the time that we are sending the message, as we can pass along the information to users that aren't meant to receive the message in order for them to act as relays for that message. The two communications mechanisms in Miraveo are the pull mechanism, in which data is published by an originator node and it can be obtained by the other nodes by a query action, and the push mechanism, where a node actively transmit data to a receiver node.

Miraveo's library is based on the Qt platform, so we are going to have to use event signalisation to be able to communicate and publish the events.

To be able to use and connect to Miraveo's network, we will have to use two main libraries: Miraveo's application library and Miraveo's network library. There are several libraries features that we are using constantly on our system's implementation, so we'll give a brief but clear explanation on how to use all Miraveo's features:

Connect to Miraveo network

The first thing we'll have to do in our application to be able to connect to Miraveo's network is to create an application instance. Miraveo uses this method to allow a single terminal to distinguish between several application instances in a device, so that messages sent to this device are uniquely directed to the application they belong to.

```
public static MVApplication* getMVApplicationInstance();
     Returns the application identifier needed to connect to
     the Miraveo network. Ownership belongs to the caller.
```

Once we have an application instance, we will be able to establish a connection by creating a network connector. This connector will be the object that will allow us to perform the actual connection to the network, linking it to the application. It will also be the object that signals all the basic events that will allow us to interact with the network: connection to the network succeeded; user connected to network and user disappeared from network.

To disconnect from the net we will just have to delete the connector's instance.

```
class MVNetworkConnector
extends QObject

Interface for interacting with Miraveo network. Signals:
userAppearedInNetwork: emitted when a new user has been
discovered userDisappearedFromNetwork: emitted when a
previously discovered user has disappeared connected: emitted
after calling MVNetwork::connect and connection attempt
finishes
```

Once connected, we are assigned a random IP address and nickname, that will be assigned to the instance of the application that we are running in that moment. Although we can select a specific nickname to use through the network, this will be prefixed by the chain &t_ for design reasons.

After having an IP and a nickname, we will be signaled every time a user appears or disappears in the network (we will see more details about that later).

Push

We are not allowed to push a DTDO anywhere in the network using Miraveo's library (well, technically we are, but we aren't achieving anything if there's no one listening to where we push). Push mechanism allows sending data to a remote node. This can only be done if a reception point, called dock, exists and is able to receive the pushed data. Every dock is identified by a tag provided by the programmer.

Miraveo's connection library offers a method that will allow us to check for docks available, so that we can know if we are able to push data in them:

```
public IMPORT_C MVNetworkConnector::RemoteQueryDocksError
queryRemoteDocks( const MVDockQueryFilter& filter, MVRemoteDocksQuery&
query );
      Starts a query to discover remote docks

Parameters:

      filter: - tag/originator filter

      query: - Dock Query Id needed to identify, obtain, read
      results and cancel the query. Query completion will be
      signaled through MVRemoteDocksQuery signal 'queryFinished'.
      Deletion of the query Object implies the query cancellation.
```

Anyway, we don't really need to know which docks are opened for every user, as in our application we already know that docks that each user has opened and what are they used for. What we actually have to do is to get the dock that we already know exists, and send the message using the push technique.

Basically we use these two functions:

```
public IMPORT_C MVRemoteDock* getRemoteDock( uint32_t tag, uint32_t
originatorIp );
      Returns a new instance of a remote dock connector.

Parameters:

      tag: - dock identifier (> 0)

      originatorIp: - remote dock originator ip (> 0) It returns
      NULL if one of precedent parameters is equal to 0 or if we
      are not yet connected to the network
```

getRemoteDock is the function that we are going to use to get the receiver's remote published dock. This way, we will be able to insert content into this dock directly, so we don't have to expect him to pull our data for us. We use different dock numbers for different ends, so we can know what kind of object we have to expect when we receive a pushed DTDO in a determined dock.

```
public IMPORT_C PushError pushDTDO( const MVLocalDTDO& dtdo, uint32_t
mpt, MVDTDOPushOperation& push );
      Pushes an unchained DTDO to this remote Dock

Parameters:

      dtdo: - The local dtdo to push

      mpt: - The maximum pushing time in milliseconds

      push: - A push operation
```

Once we already have the remote dock object, we can freely push our DTDO's to it by using **pushDTDO**. The DTDO can be anything we want it to be, so we have to know beforehand what the user is expecting to receive in that particular dock, and sent the data accordingly.

Before we can push a DTDO into a desired dock, the user we are trying to connect with must have previously created this dock, offering a standpoint to which other users can send data in form of DTDOs.

```
public IMPORT_C MVNetworkConnector::CreateDockError createLocalDock(
MVLocalDock& dock, uint32_t tag );
      Creates a new DOCK

Parameters:

      dock: - dock object

      tag: - Local Dock public identifier
```

createLocalDock is really simple in its form in the sense that we only have to provide an MVLocalDock object and a tag. This local dock will signal us every time we have new objects in our dock. We will have to create handlers that manage these DTOs, but we will talk about this later, in the application's functions details.

Pull

Pull technique is, as we know, conceptually different than push technique in the sense that we don't have to expect other users to send us data, but we have to proactively select what data we want and ask for it if it is available.

To do so, we will have to bear in mind two concepts: Publishing a DTDO and requesting a published DTDO.

To publish a DTDO we just have to use Miraveo's library to do so. We must create the object we want to publish and create a DTDO with it (we'll see how later). Once we have created the DTDO object, we can safely publish the DTDO using any tag number we want to use.

```
public IMPORT_C MVNetworkConnector::PublishDTDOError publishDTDO( const
MVLocalDTDO& dtdoId );
      Publishes the DTDO identified by dtdoId. After this call
      remote users will be able to access to the publish DTDO.

Parameters:

      dtdoId: - DTDO identifier
```

We can, in a similar way, unpublish a DTDO that has already served its purpose:

```
public IMPORT_C MVNetworkConnector::UnpublishDTDOError unpublishDTDO(
const MVLocalDTDO& dtdoId );
      Unpublishes the DTDO identified by dtdoId. After this call
      the DTDO will not be accessible unless it is published again.

Parameters:

      dtdoId: - DTDO identifier
```

DTDO requesting. They are both asynchronous methods that will need Qt signalizations implemented in order for them o fulfill their goal.

DTDO querying will allow us to know which DTDOs are available among all users in the network. This method in intended to give us an idea of what DTDOs are in that moment available, to let us chose which ones of them do we want to actually request:

```
public IMPORT_C MVNetworkConnector::QueryDTDOsError queryDTDOs( const
MVDTDOQueryFilter& filter, MVDTDOQuery& query );
      Throws a query for DTDOs. It does not return DTDO data but
      just existing Dtdos' identifiers.

Parameters:

      filter: - tag/originator/seeder filter

      query: - DTDO Query Id needed to identify, obtain, read
      results and cancel the query. Query completion will be
      signaled through MVDTDOQuery signal 'queryFinished'. Deletion
      of the query Object implies the query cancellation.
```

DTDO Requesting will, on the other hand, allow us not only to request identifiers from DTDOs, but also will get their data. We can combine both methods so that we know which data do we want to pull every time we are requesting a DTDO, or directly request all of them and work on the fly.

```
public IMPORT_C MVNetworkConnector::RequestDTDOsError requestDTDOs( const
MVDTDOQueryFilter& filter, uint32_t mrt, MVDTDORequest& request );
      Throws a request for DTDOs.

Parameters:

      filter: - tag/originator/seeder filter

      mrt: - the Maximum Retrieval Time is the maximum amount of
      time in miliseconds during which MLib will try to obtain the
      requested DTDO/s (it may last less than MRT value).

      request: - DTDO Request Id needed to identify, obtain, read
      results and cancel the request. Query completion will be
      signaled through MVDTDORequest signal 'requestFinished'.
      Deletion of the request Object implies the request
```

Both methods can filter the search results we want to acquire by using the filter object. In the filter, we can:

- Select the originator's IP: Setting an originator's IP will allow us to filter the messages in a way that only those that have been created by a specific IP will be the ones that will be delivered to us.
- Select the seeder's IP: This will allow us to select a specific address as a seeder. This means that the messages that we will get will be the ones that reside in him, being himself the originator or not.
- Select the tag number.

This way, we can easily discriminate which messages we want to receive.

In both methods we will receive signals that will indicate when the requests have finished so that we are able to read the results.

Create DTDO

The objects that we are going to create and distribute over the network using the techniques we have seen are called DTDOs (Delay Toleran Distributed Objects). Using Miraveo's library, we can create DTDO's and publish them in a way that other users can obtain them. The key feature about DTDOs is that they can exist beyond the disconnection of they're creator. This way, we are able to create a system that can distribute this objects to their destiny although the original sender and the destiny user have never discovered each other in the network at the same time.

To create a DTDO we use Miraveo's network connector class:

```
public IMPORT_C MVNetworkConnector::CreateDTDOError createDTDO(
MVLocalDTDO*& dtdo, uint32_t tag, const QByteArray& data, int32_t ttl );
      Creates a local DTDO Object

Parameters:

      dtdo: - identifier of the created DTDO to be returned.
      Ownership belongs to the caller.

      tag: - identifier tag (must be < MAX_DTDO_TAG)

      data: - data of the object. Size in bytes must be 1 <=
      data.size() <= MAXDATASIZE_DTDO

      ttl: - amount of time in miliseconds the created DTDO is
      accessible outside the origin node (MINTTL_DTDO <= ttl <=
      MAXTTL_DTDO)
```

Using this call will provide us with a DTDO that will be identified witch a tag that we are giving to it. Miraveo's network identifies unique DTDO by both they're originator's IP and the tag they have assigned. This way we don't have to worry if another user has created a DTDO that has the same tag as we did.

Another useful feature of the DTDO creation is the *ttl* parameter. This parameter represents the Time To Live for the DTDO that we created, and represents the time that this object will be alive on the network once outside the original sender. This way we can assure that the message will not be out in the network indefinitely, and that it can only be requested for a determined amount of time.

The dtdo parameter will contain the created object that will actually have the DTDO and will allow us to publish it or push it in any way we want.

Development guide

In this section we will go through implementation details on how we handle the main network functions used repeatedly in the application. This way, we'll illustrate how we are using Miraveo's library to create and populate the network.

Connecting to the network

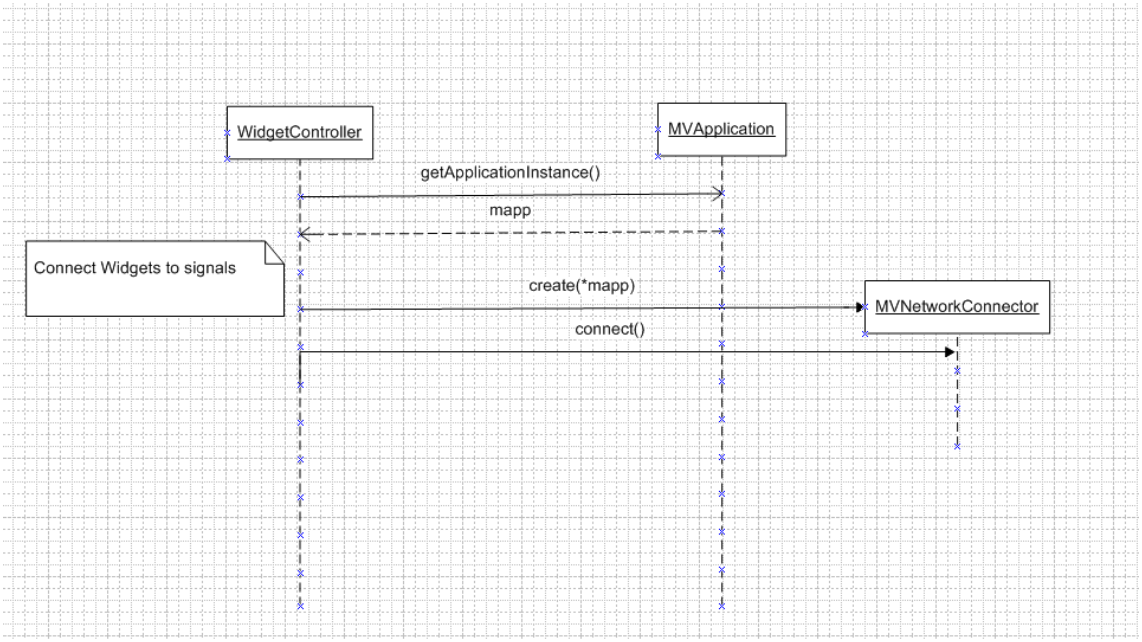
When a user connects we have to:

a) Ensure that he connects to Miraveo's network
b) Provide the methods to let him freely exchange data with the other users of the network.

To do so, we'll use the calls that we have already seen in Miraveo's library summary, following Miraveo's paradigm to ensure a good usage of the network. In a nutshell, to perform a) we should:

1. Create a Miraveo application instance
2. Create a Miraveo network connector based on that instance
3. Connect the signals sent by the connector object to some slots we created locally. We have to specifically connect the signals that are sent when a user connects to the network, when a user disconnects from the network, and for when the connection is completed.
4. Create a local User that has the nick that we use on the *ConnectWidget*.
5. Use that user to connect to Miraveo's network.

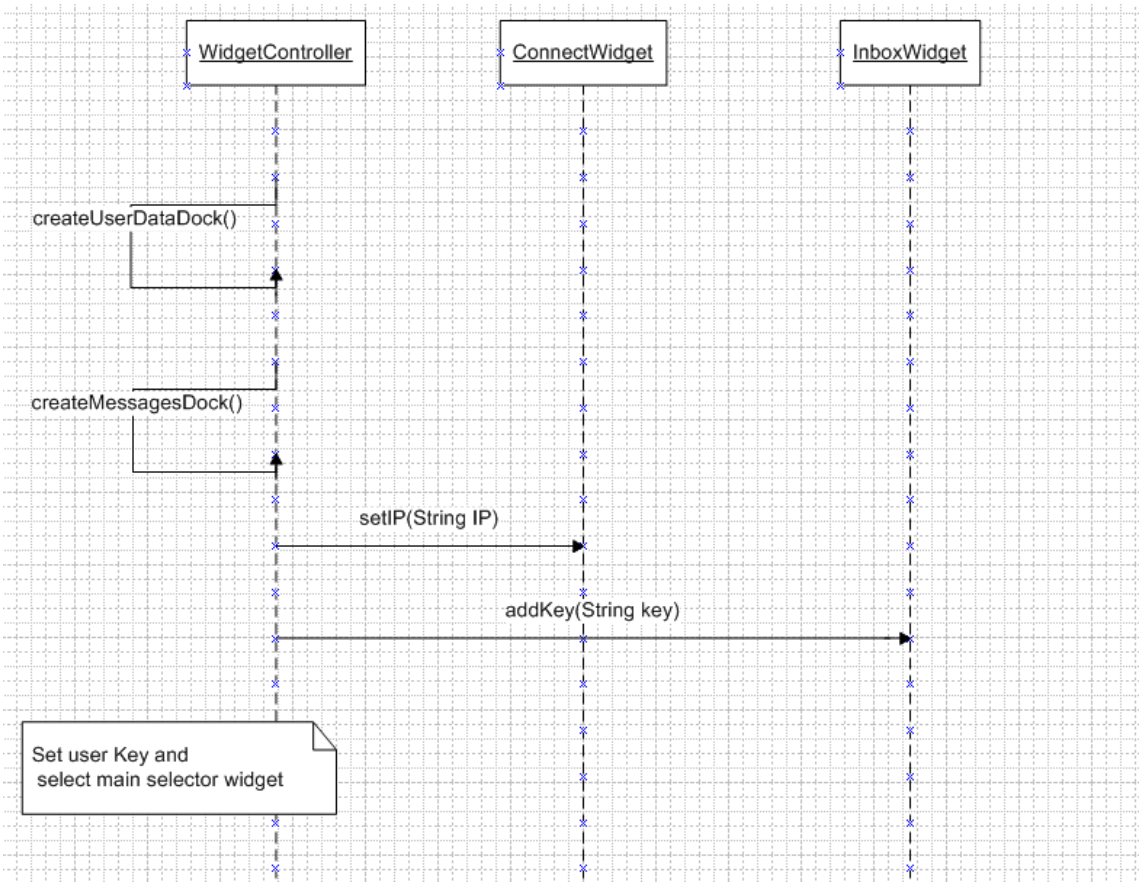
```
MVApplication* mapp = MVApplication::getMVApplicationInstance(); // Miraveo Application declaration
iMConn = new MVNetworkConnector(*mapp); //  Creating the connector


/*  Connecting signals:
QObject::connect(iMConn, SIGNAL(userAppearedInNetwork(const MVUser &)),
        this, SLOT(userAppearedInNetwork(const MVUser &)));
QObject::connect(iMConn,
        SIGNAL(userDisappearedFromNetwork(const MVUser &)), this,
        SLOT(userDisappearedFromNetwork(const MVUser &)));
QObject::connect(iMConn,
        SIGNAL(connected(MVNetwork::MVNetworkConnectionError)), this,
        SLOT(connected(MVNetwork::MVNetworkConnectionError)));

//  Connection to the Miraveo Network is started
MVLocalUser luser(connectWidget->nick);
MVNetwork::MVNetworkConnectionError err = MVNetwork::connect(luser,
        *iMConn);
```



Once we've created the connection, we must wait for it to establish. If it is successful, we can go on and prepare the application to receive user notifications and messages by:

1. Creating a dock for receiving user information.
2. Creating a dock for receiving messages.
3. Update the variables with the user and connection information
4. Add the connected user key to the list of keys that we will be receiving.
5. Set the main selector screen as the one we'll be showing to the user.

```
MVMessageBox::information("Connected to Miraveo");
//create a dock with tag 1 that will receive user info
this->doCreateDock();
//create a dock to receive messages
this->doCreateUserDock(3);
connectWidget->user.ip = iMConn->myself()->integerIp();
inboxWidget->keys.insert(connectWidget->user.key.toInt(),
        connectWidget->user.key);
this->myKey = connectWidget->user.key;
setMainSelectorCurrent();
break;
```



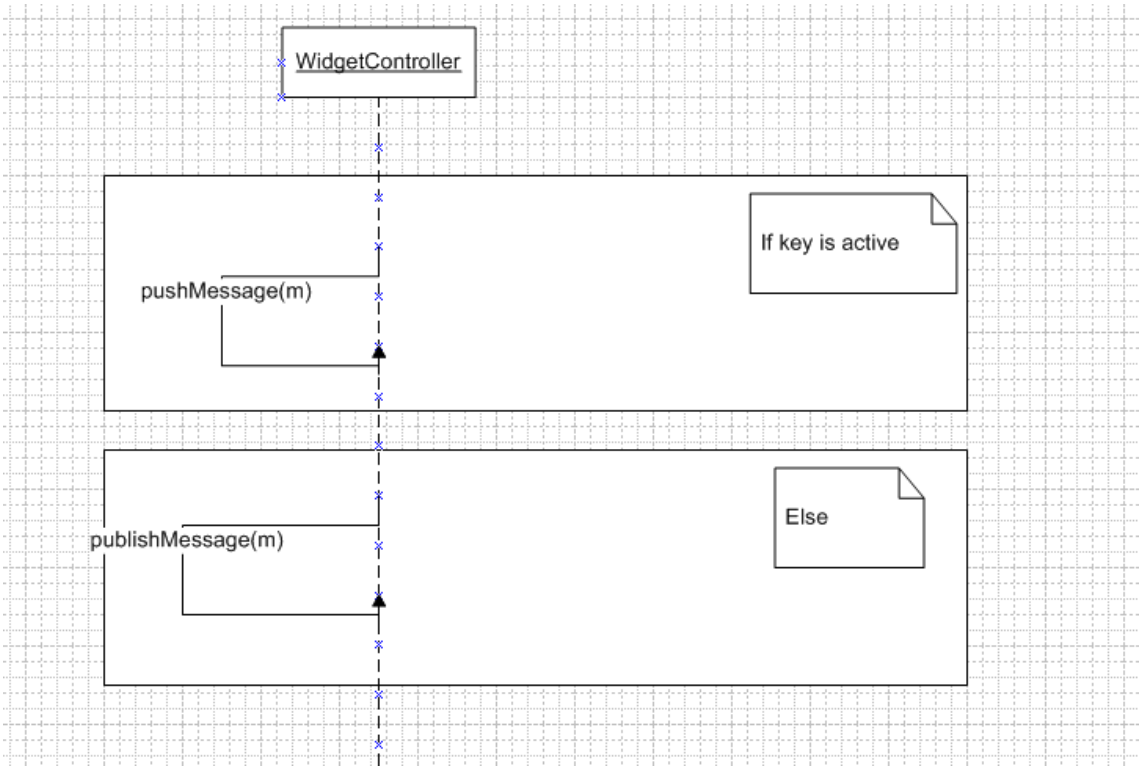
Once we've done all this, we'll be up and ready to start using the application.

Sending a message

To send a message, we have to discover whether we can send it directly by pushing it into a dock or we can just publish it and hope that the message will arrive to its destiny. When we are sending messages to users we can do that in a relatively easy way, as we keep track of which users are connected at every moment.

When sending messages to groups, though, we always publish them, as we have no idea of who is a member of a determined group.

```
void WidgetController::sendMessageDTDO(Message m)
    {
    if (cUsersKeys.contains(m.keyDest))
        {
        MVMessageBox::information("Pushing message");
        m.ip = QString::number(cUsersKeys.value(m.keyDest));
        this->sendPushMessageDTDO(m);
        }
    else
        {
        MVMessageBox::information("Message is going to be published");
        this->doMessagePublish(m);
        }
    }
```

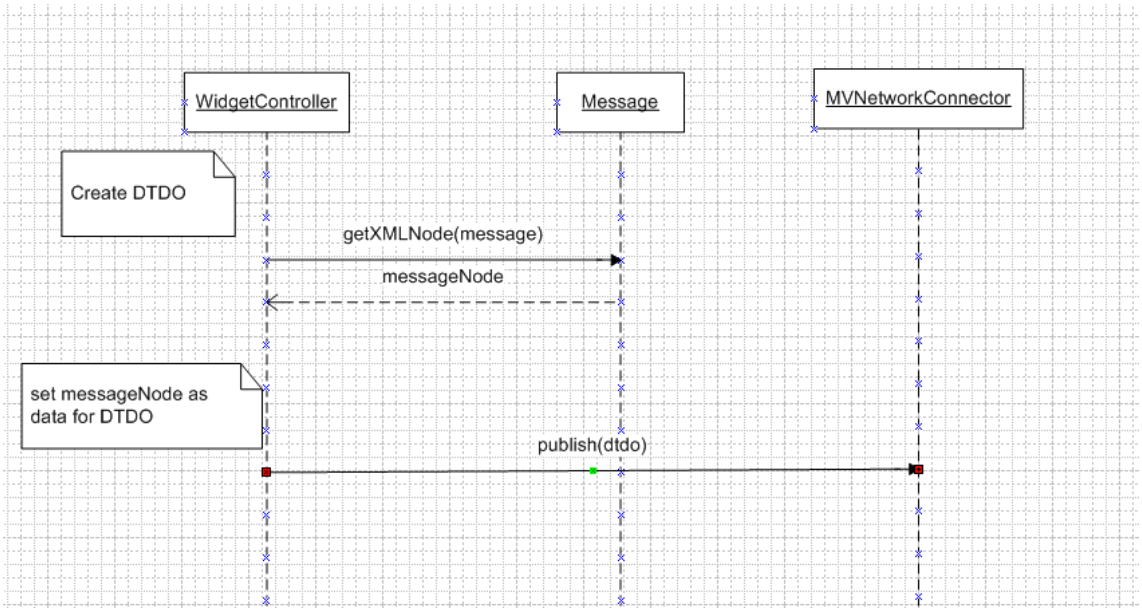


As we can see, we:

1. Check if the user's key matches any of the connected IPs.
2. If the user key is found, we add the IP to the message and push the message to the destiny. If it's a group message, this key will never be found as it is never added.
3. If it doesn't, we publish the message and expect the receiver to proactively ask for the message.

In the case that we publish the message, we follow Miraveo's paradigm and create a DTDO object that we will publish with a randomly generated tag. We generate a random tag because we use the user's key to identify them, so we just want to avoid a conflict.

```
MVLocalDTDO* dtdo;
QDomElement d;
QDomDocument doc;
d = m.createXMLNode(doc);
doc.appendChild(d);
QByteArray bdata;
bdata.append(doc.toByteArray());
// DTDO creation
QTime time = QTime::currentTime();
qsrand((uint) time.msec());

// Get random value between 0-65536
int randomValue = connectWidget->randInt(0, 65536);
MVNetworkConnector::CreateDTDOError crErr = iMConn->createDTDO(dtdo,
        (uint32_t) randomValue, bdata, 300000);
if (crErr == MVNetworkConnector::CreateDTDONoError)
    {
    // DTDO publication
    MVNetworkConnector::PublishDTDOError pbErr = iMConn->publishDTDO(*dtdo);
```



As we can see, when we publish a message, we:

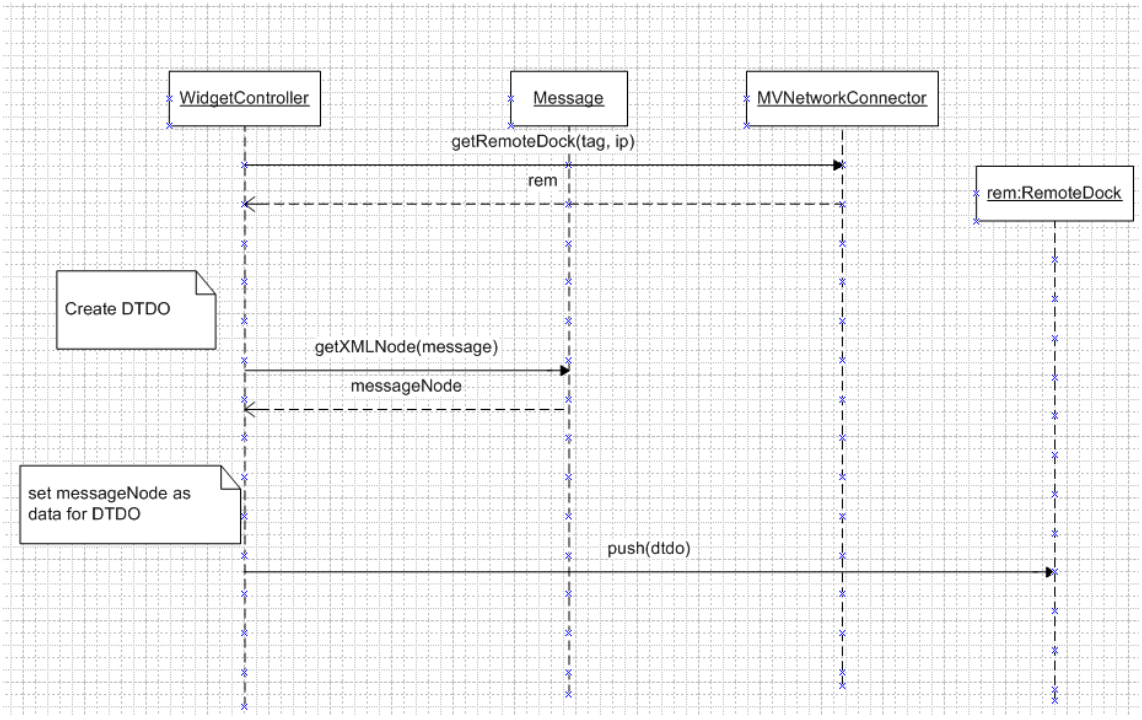
1. Create a Local DTDO object.
2. Create a new DomElement from the message that is meant to be sent.

3. We convert that DomElement into a byte array so that we can send it easily through the network.
4. We create a random value to add a tag to the DTDO
5. We create the DTDO using Miraveo's network connector.
6. We publish the DTDO

After that, if no error is found, the DTDO is ready and published in the network.

Our procedure to push the DTDO is obviously different both in ways and in philosophy.

```
// Geting the dock where to push
iRemoteDock = iMConn->getRemoteDock(tag, m.ip.toInt());
if (!iRemoteDock)
    {
    MVMessageBox::error("Impossible error: getRemoteDock wrong parameters");
    return;
    }

// Creating the operation to listen to
iPushOp = new MVDTDOPushOperation();
QObject::connect(iPushOp, SIGNAL(pushFinished()), this,
        SLOT(pushFinished()));
QDomElement d;
QDomDocument doc;
d = m.createXMLNode(doc);
doc.appendChild(d);
QByteArray bdata;
bdata.append(doc.toByteArray());

// Pushing
MVDTDOPusher::PushError err = iRemoteDock->pushData(bdata, MIN_MPT,
        *iPushOp);
```



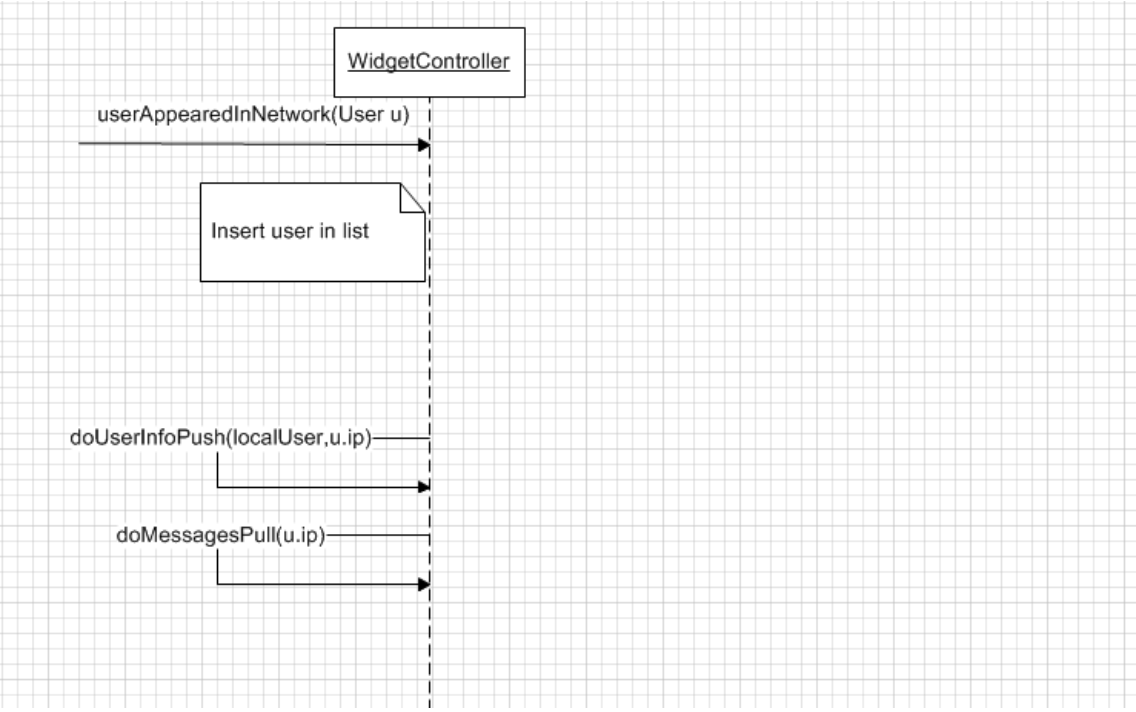
This time we are:

1. Getting the remote dock where to push the message: we already know the dock we are pushing to.
2. Creating a new Push Operation using the library.
3. Connecting the signal sent when the push operation is finished to a local slot. We use this slot to detect if there has been any error while sending the message.
4. Creating a new DomElement from the message that is meant to be sent.
5. Converting that DomElement into a byte array so that we can send it easily through the network.
6. Pushing the message to the desired destiny.

User connected

When a user is connected, we have to do several things in order for us to maintain a coherent and up-to-date network.

```
void WidgetController::userAppearedInNetwork(const MVUser &user)
    {
    iConnectedUsers++;
    MVUser *new_user = new MVUser(user);
    iUsers.insert(std::make_pair(user.integerIp(), new_user));
    //We push our user data to the new user
    doUserInfoPush(connectWidget->user,user.integerIp());
    //We get the messages we are interested at
    doMessagesPull(user.integerIp());
    }
```

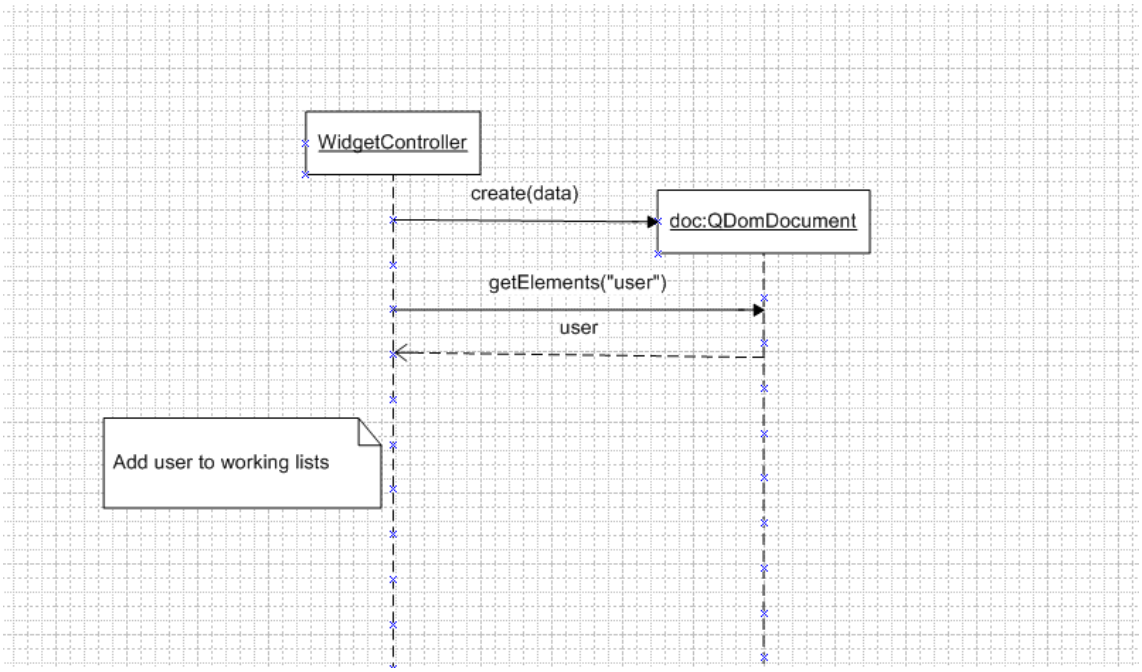


Any time we detect a connection, we must:

1. Insert the user into the connected user list.
2. Update the user count.
3. Push our information into the connected user's open dock.
4. Get the messages we want from the user

The information that we pass to the user is used to map the correct user info. Therefore, there's one more process we must do when a user is connected: parse the information the connected user sends us by pushing it to our open dock. The code that we use to parse the user looks like this:

```
//We create a user to be parsed
CUser c;
QByteArray ba((char*) it->data, it->dataSize);
QDomDocument doc;
if (!doc.setContent(ba))
    MVMessageBox::information("unable to parse dtdo");
else
    {
    //We search for the element "user" in the DomElement
    QDomNodeList elements = doc.elementsByTagName("user");
    for (int i = 0; i < elements.size(); i++)
        {
        QDomElement q = elements.at(i).toElement();
        //Parsing the element
        c.key = q.attribute("key");
        c.nick = q.attribute("nick");
        c.password = q.attribute("password");
        c.ip = q.attribute("ip").toInt();
        cUsers.insert(c.ip, c);
        //Mapping of the key with the ip
        cUsersKeys.insert(c.key, c.ip);
        //We set the list in the network users screen
        nuWidget->setList(cUsers);
        }
    }
```



What we do when we get a user's info DTDO is:

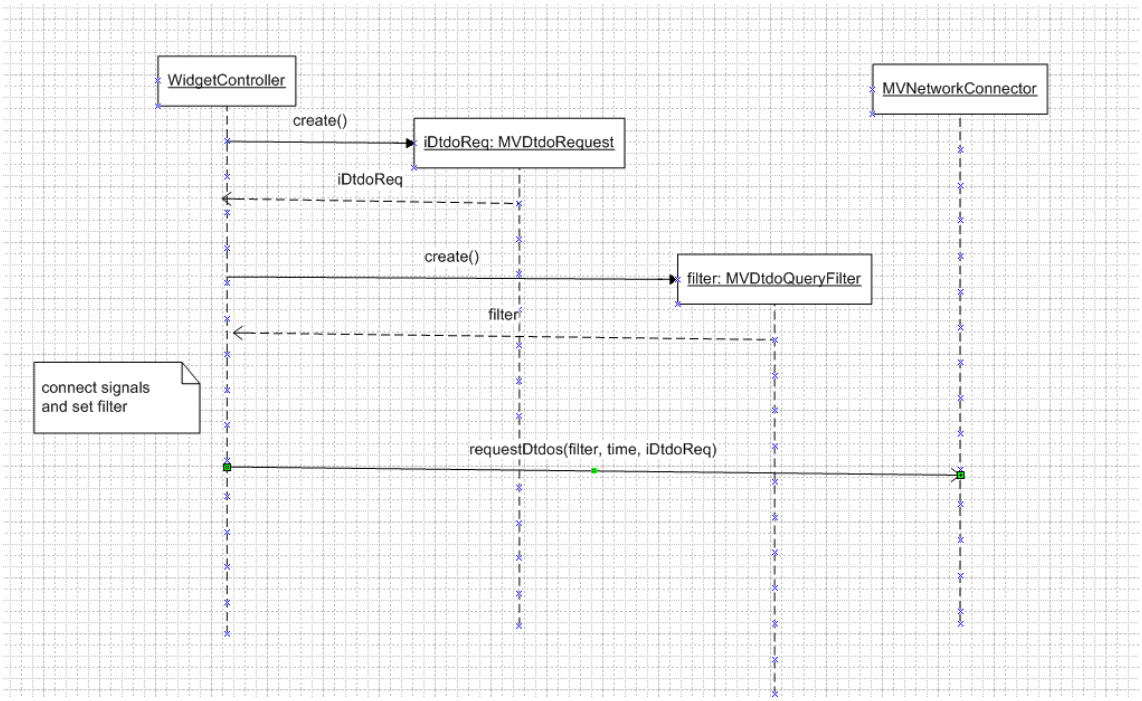
1. Parse the DTDO's data into a DomDocument.
2. Search for the tag "user" inside the DomDocument. This way, we'll have the element that has the user's information.
3. Parse the user into an object and set it into the user's list.
4. Map the user's key with his IP, so that we can always be aware of what users can receive pushed data.

Once we've done all this, all that's left is the processing of the user's messages. When we do this, what we are doing is to select which actions we want to take for the message routing. Here is an example of a simple message routing in which we demand all messages available,

but the policy is completely up to us. I've maintained this simple so that several experiments and tests can be done by just changing this policy. The steps we follow are:

1. We create a new DTDO request.
2. We connect the signal that is going to be sent when the messages are received with the appropriate slot. We will talk about processing received messages later.
3. We set the filter to whatever we want. In this example, we are interested in all the messages that the new user has to offer, both created and seeded by him.

```
void WidgetController::doMessagesPull(uint32_t orig)
    {

    if (!iMConn)
        {
        MVMessageBox::error("Not connected");
        return;
        }

    if (iDtdoReq)
        {
        MVMessageBox::error("Pending request");
        return;
        }

    // We create a new request
    iDtdoReq = new MVDTDORequest();

    QObject::connect(iDtdoReq, SIGNAL(requestFinished()), this,
            SLOT(processRequestMessagesFinished()));
    MVDTDOQueryFilter *filter = new MVDTDOQueryFilter();
    //Now we are getting all the messages that are being either generated by the user or seeded by him
    filter->setSeederIp(orig);

    MVNetworkConnector::RequestDTDOsError reqErr = iMConn->requestDTDOs(
            *filter, 1000, *iDtdoReq);

    if (reqErr != MVNetworkConnector::RequestDTDONoError) //    If error operation is aborted
        {
        delete iDtdoReq;
        iDtdoReq = NULL;
        MVMessageBox::error("Request failed");
        }

    }
```


WidgetController
MVNetworkConnector
create()
iDtdoReq: MVDtdoRequest
iDtdoReq
create()
filter: MVDtdoQueryFilter
filter
connect signals
and set filter
requestDtdos(filter, time, iDtdoReq)

Adding or deleting a friend

Adding and deleting a friend of our own is relatively simple. We just have to make sure that we create the appropriate structure in the file that holds all the friends structure and append it accordingly. We won't add any code snippets here because the code is quite irrelevant to the network.

Adding or deleting a group

This case is exactly the same as the "Adding or deleting a friend" case. In a similar way, we won't be posting the code here as isn't relevant or interesting.

User disconnected

When a user disconnects, we receive the appropriate signal from the library. The procedure we follow is just to make sure that we make him disappear from all the lists that have the user's information, so that we have a clear picture of the network and also to make sure that the appropriate networks don't show the user's information anymore.

```
void WidgetController::userDisappearedFromNetwork(const MVUser &user)
    {
    MVMessageBox::information("User disappeared");
    iConnectedUsers--;

    std::map<uint32_t, MVUser*>::iterator it = iUsers.find(user.integerIp());

    if (it != iUsers.end())
        {
        delete it->second; //   delete of the second element is necessary because it is a pointer!
        iUsers.erase(it);
        }

    QMap<uint32_t, CUser>::iterator it2 = cUsers.find(user.integerIp());
    if (it2 != cUsers.end())
        {
        cUsersKeys.remove(it2->key);
        cUsers.remove(user.integerIp());

        }

    }
```

As it is a very basic case, we will not be adding a UML sequence.

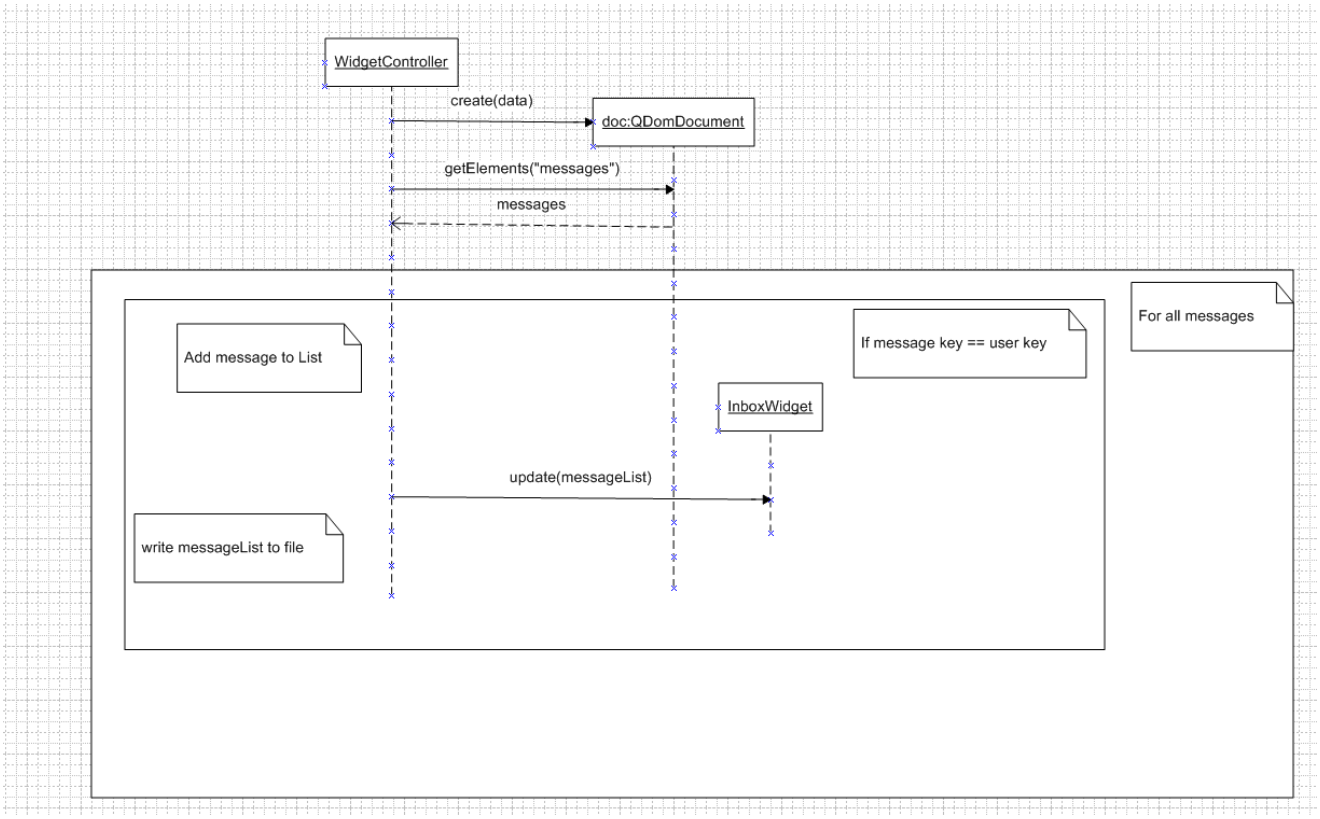
Receiving a message

Receiving a message is a relatively simple matter. What we have to do when we receive one is:

1. Parse the message in a way that we can get the data that we are interested in from it.
2. Check if the message is specifically destined to us or one the groups we are interested in.
3. If the message is destined to us or any of our groups we must:
   3.1. Add the message to the message list that we will convert into a file.
   3.2. Update the inbox
   3.3. Write the message so that we can check it anytime

It really doesn't get any different whether it is a pushed message or a messaged that we've looked for; the only difference is the way we've arrived to retrieve the message.

```
//Convert the data to a Document
QByteArray ba((char*) it->data, it->dataSize);
if (!doc.setContent(ba))
    MVMessageBox::information("unable to parse dtdo");
//Look for the found messages
QDomNodeList elements = doc.elementsByTagName("message");
for (int i = 0; i < elements.size(); i++)
    {
    q = elements.item(i).toElement();

    m.ip = q.attribute("ip");
    m.keyOrig = q.attribute("keyOrig");
    m.keyDest = q.attribute("keyDest");
    m.body = q.attribute("body");
    m.nickOrig = q.attribute("nickOrig");
    m.nickDest = q.attribute("nickDest");
    m.subject = q.attribute("subject");
    m.timestamp = QDateTime::fromString(q.attribute(
            "timestamp"));
    m.id = q.attribute("id");
    //We are only interested in keeping the message if the key that's embedded in it is ours
    if (myKey == m.keyDest)
        {
        if (messageList.contains(m.id))
            {
            if (messageList.value(m.id) != m.keyOrig)
                {
                //Append the message to the list that we write to a file
                rootDest.appendChild(m.createXMLNode(
                        docDest));
                }
            }
        else
            {
            //Append the message to the list that we write to a file
            rootDest.appendChild(m.createXMLNode(docDest));
            }
        }

    .
//We also want our group messages
else if (groupKeys.contains(m.keyDest))
    {
    if (messageList.contains(m.id))
        {
        if (messageList.value(m.id) != m.keyOrig)
            {
            rootDest.appendChild(m.createXMLNode(
                    docDest));
            }
        }
    else
        {
        rootDest.appendChild(m.createXMLNode(docDest));
        }
    }

}
```


WidgetController
create(data)
doc:QDomDocument
getElements("messages")
messages
For all messages
Add message to List
If message key == user key
InboxWidget
update(messageList)
write messageList to file

Conclusions

Creating a delay tolerant messaging application wasn't easy. It wasn't for various reasons: Because of the background one should achieve before starting to create the application, because it was meant to be an application that would be easily modifiable and, last but definitely not least, because it sometimes could be a difficult task to program for Symbian.

The main issue I encountered was that debugging and error was, in the best scenario, difficult. Qt is supposed to have an S60 emulator that should work on a windows platform, but it seems to work only on specific computers, and mine was not one of those. Talking with the team that developed the Miraveo library, I encountered that they had the same exact issues when they were developing they platform, so it was not something new.

Another main stopper was Qt's debugging service. It simply didn't work for me, and it also seems to be an extended issue. The problem is that, dealing with pointers, complex data structures and a library that almost no-one had worked with, it was a titanic effort to find out what were the issues that were causing issues when they were well-hidden. In another system, simply with a debug-friendly environment, issues that would take me weeks could have just take a day or two.

There's nothing to do to solve this problems, and once one is quite familiar with Qt, Symbian and Miraveo's library you get to know the issues quite fast and have a fast reaction, but that may be a stopper for most people that want to develop an application using the library.

Another issue that I had was the fact that Miraveo's API needed quite a bit more of work. This was an issue until they sent me some sample applications that helped me understand much better how they handled most of their capabilities.

On the bright side, I believe it has been quite an interesting project. It changed a lot over time, as it started as an attempt to analyze Miraveo's deployment data, but I believed it has changed for better. Now the research group doesn't have to rely on a third party to develop applications for them or provide them with the information they need, as they have their own application that they can understand and tamper all they want with.

There have been major changes during the development of this application. The use of xml files as indexes for all data was a painful but necessary change, as the objective of the application was, precisely, to have another person come over later and be able to perform some testing runs using it and would, therefore, need to change it. It came with a major refactor of the code, but I believe it was for better, as it also gave me quite a broad view of the library once I had to deal with the coding for a second time.

Overall, I believe the application has met the expectations, and it will be an asset for the research group in a near future.

Future approaches

Now that the application base is done, the research group will have to prepare and deploy an experiment, that can involve or not third-party users as students, UPC staff or others. They may have to bear in mind certain constraints that can limit the usability:

- Battery constraints have proven to be an issue in all implementations.
- Application has to be robust, they need to do several test runs for their logging, expected results, etc.
- It is difficult to have a solid user base, as there has to be a big network activity for users to be interested.

Overall, I would say that an endemic weakness of this type of application is that it is only interesting to connect to it when there is a big user base to join at, and this user base can't be created artificially. As we have seen in the similar approaches that other universities have driven, we have to be very cautious about user population.

It would be interesting not only to log and monitor usual tests as inter-contact time, data rate, error rate, etc. but also take a look at MIT's testing, and try to approach the testing into areas as social groups detection, heuristic routing for messages, etc.

For instance, if we detected that a group of people are family or friends, and tend to meet often, and another group of people have never seen them, it may make sense only to forward the message to those that would most probably get in the touch with the destiny. We could transfer this heuristic data in a relatively easy way in the application when we send our own user data.

It will be interesting to change the application once the first testing results have come, as that s what it mainly lacks: a field perspective.

Cost Analysis

In this project there was only one person involved. The total cost of it represents the total cost of the hardware used, the software purchased and the human time costs.

Hardware costs

The hardware used for this project was:

- Three Nokia 5530 smartphones, at the cost of 200€ each.
- A PC, for human work, that was valued in 1200€

Software costs

- There are no software costs in this project, as all the software involved was free.

Human costs

Human costs in this project are divided in the work hours.

- Study of basic in ad-hoc networking: 80 hours
- Testing of Miraveo's library: 80 hours
- Start developin in Qt framework: 50 hours
- Application Design: 100 hours
- Application Development: 200 hours
- Application Debugging: 200 hours
- Memory development: 50 hours

Calculating a cost of 10€ per hours as a temptative cost, the cost in human hours has been:

760 hours * 10€/hours = 7600€

Total costs

Total costs of this project represent the addition of all costs, resulting in a total of:

7600€ + 1800€ = 9400€.