Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

MASTER THESIS

Implementation of a Disaster Mode to Maintain Twitter Communications in Times of Network Outages

Paolo Carta

Advisors: Theus Hossmann, Dr. Franck Legendre

Supervisors: Prof. Dr. Bernhard Plattner, Prof. Dr. Daniele Giusto

Thesis submitted as part of the M.sc degree

in the Department of Telecommunication Engineering,

Cagliari (Italy)

August 2011

Abstract

Online social networks like Twitter and Facebook have become critical infrastructure for communicating and spreading news, especially in times of natural disasters or political uprisings. To date, these social networks depend on wired and wireless network infrastructure. This dependency is dangerous, as challenged governments can (as the recent example in Egypt shows) shut down the network to disrupt communication. Network infrastructure is also susceptible to natural disasters like strong earthquakes or floods. In the last few years smartphones and applications for them have become extremely popular and the growth is supposed to increase in the next years. Such devices have the capability to communicate wirelessly via Bluetooth and sometimes WiFi Ad-hoc without relying on existing network infrastructure. This gives the possibility to exchange data opportunistically in a epidemic way.

In this master thesis a disaster ready application for Android phones (Twimight) has been implemented. Beyond the basic Twitter functionalities it gives the user the possibility to exchange Twitter messages (tweets) directly among mobile devices without necessarily having access to the WiFi in infrastructure mode or 3G. A real life experiment has been carried out to evaluate the performance of the client and some interesting results have been produced, namely Delivery Delay CDF, Hop Count, Delivery Ratio and some Battery consumption statistics. The security aspects of this architecture are discussed. In addition, Twimight is proposed as a platform to automatically disseminate sensor readings that can help in the aftermath of a disaster. They can be translated in fact into tweets without any user interaction. They can give additional information about the user's environment as well as providing a sign of life. The sensors used are the ones available on the Android mobile phone, namely accelerometers, GPS sensor, light sensor. The combination of all the features makes this opportunistic Twitter client the ideal emergency kit in situations of disasters.

Alla mia famiglia

Acknowledgments

I would like to thank:

- Professor Bernhard Plattner and Dr. Franck Legendre that offered me the opportunity to write my master thesis at the Communication System Group of the ETH in Zurich.
- Theus Hossmann for his continuous help and feedback during the whole period of this thesis.
- Professor Daniele Giusto for giving me the possibility to study at the ETH Zurich during this academic year.
- My family, since without their love and encouragement I would have never managed to achieve this goal .
- My colleague, Abdullah Alhussainy for the ideas shared together and the sleepless nights during these months.

Contents

List of Figures

List of Tables

1. Introduction

Platforms like Facebook and Twitter have become really popular during the past few years. They improved significantly the way of sharing and spreading information and revealed themselves particularly suited in case of disastrous events such as tornadoes, earthquakes or even political uprisings [11]. They provide in fact the possibility to the users to contact each other, to create groups and share any type of content rapidly from multiple platforms.
Their increasing importance combined with broadband flat rate Internet access and the diffusion of smartphones to the public has been a revolution in terms of possibility of access information everywhere at any time at speeds never possible before.
However whether the access to these platforms is performed using fixed traditional networks or mobile ones they are always characterized by the same drawback: they depend on wired or wireless infrastructure. They were not designed to support emergency situations and hence lack of some features that might improve their effectiveness in such events. The dependency from infrastructure is therefore pretty dangerous since there might be situations in which connection is not available. Incidentally the events in which social networks and micro-blogging services are most needed (i.e. natural disasters or revolutions) are the ones in which network outages might occur both intentionally or not.
In order to solve these problems, wireless mesh networks [12] and satellite communications have been proposed . Unfortunately they both need to be deployed first and are not rapidly available. Today smartphones however have the capability to communicate among each other with Bluetooth and WiFi ad-hoc (even though not always supported by default). They might be exploited therefore in order to spread information opportunistically and support social networking applications, for instance even in remote areas or during disasters. Opportunistic architectures have already been proposed, such as Podnet or Haggle. The former is too specific since it is designed mainly for distributing podcasts, the latter is a bit too generic since it is an architecture for spreading information within ad hoc networks made up by Windows Mobile phones. Therefore, we decided to start from scratch in order to have a simple and effective way to spread tweets among mobile phones. Moreover opportunistic networks can increase capacity [13], and offload the 3G network infrastructure [14] and use the spectrum more efficiently [15].
An innovative disaster ready Twitter application for Android phones called Twimight was developed. It allows the user to use his own Twitter account as well as provide opportunistic spreading of information when the disaster mode is active. In this case in fact the device will work in a "Hybrid" way in which information is sent via Bluetooth as well as on the central servers when connectivity is available. The number of new features provided is limited to avoid any additional confusion for the users in disastrous events. The most important are sending a tweet, reply or send a direct message to a specific user. In this way the transition from the normal centralized Twitter to the delay-tolerant one is as seamless as possible.

The performance of client were tested in a typical office scenario. It involved 5 people, each one had to carry the device during the day of normal working. Some statistics were collected and used to produce some plots such as Delivery Delay CDF, Hop Count, Delivery Ration and some Battery consumption statistics. The security design choices and the final foreseen architecture are explained. The main assumption for the architecture is that the security elements must be set up when the clients authenticate and therefore before a real disaster occurs. When a disaster takes place Twimight has to be able to work in a complete distributed way. Finally the application is proposed as a platform in order to publish online sensor data reading that can provide useful information in case of disasters. The reading coming from the internal phones sensors can be easily translated into tweets in order to give additional information about the user's environment as well as automatically providing a sign of life. The sensors used are namely, accelerometers, GPS sensor, light sensor.
Summarizing the main **contributions** of this thesis are:

- Implementing of Twimight
  - Fully functional Twitter client for Android phones
  - Disaster mode to spread data opportunistically
  - Extensible with support for "plugins"
- Experiment Twimight in a real world scenario
- Raise awareness of the importance of Wifi ad-hoc and Delay Tolerant Networks.
- Introduce Twimight as a platform for spreading sensor data readings in cooperation with the Uppsala University.
- Made work public available
  - Code available in an online repository
  - App will be published on the market soon
  - A paper has been submitted and accepted at the ExtremCom 2011, 3rd Extreme Conference on Communication - The Amazon Expedition.
  - Project will be presented at the CoNEXT 2011: Workshop on Disaster, Japan.

1.1. Outline

The next chapter presents some essential background. It explains some scenarios in which the application could be used. Chapter 3 discusses Related work. It gives a brief background of PodNet ,Haggle and Mobiclique and some other work related to this thesis. Chapter 4 discusses the most important open source Twitter clients available online. The architecture and design of the application are described in chapter 5. The real life experiment of the application, how it was carried out and its results are written in chapter 6. Chapter 7 discusses the security aspects of Twimight. Chapter 8 explains how sensor readings can be spreaded on Twitter. Finally there is a summary of this master thesis in chapter 9.

2. Background

This chapter explains why an opportunistic extension of Twitter might help when network infrastructure is down. First some statistics and numbers about Twitter are presented, then some recent cases in which it has been heavily used are discussed. Moreover the need for an alternative way to communicate and use social networks is pointed out.

2.1. Why Twitter

It is a popular micro-blogging service which has seen an incredible growth in the past few years. Up to date (May 2011) it represents a community of about 200 million users as shown in the figure below:

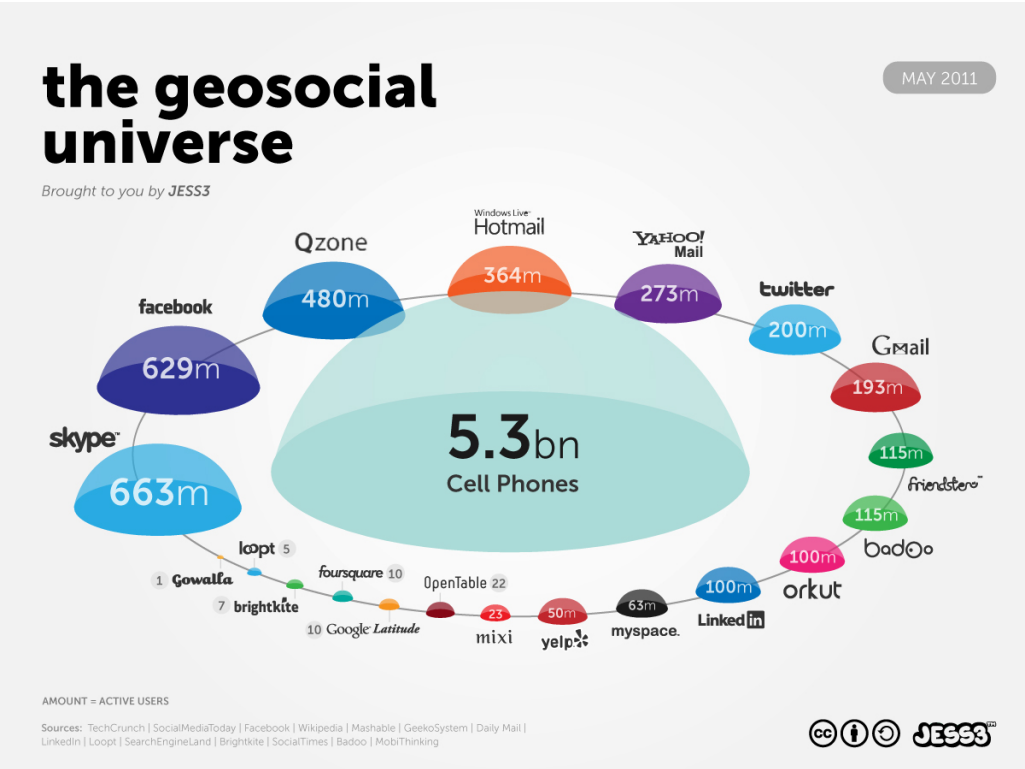


Figure 2.1.: Social Universe Image [1]

If we just consider normal online social networks, Twitter represents the second most used after Facebook. Its growth in the last few years has been huge. This can be easily seen in figure 2.2. Users were tweeting 5,000 times a day in 2007. By 2009 it had grown to 2.5 million per day. Today, we are seeing around 200 million tweets per day and the increase in number of mobile users

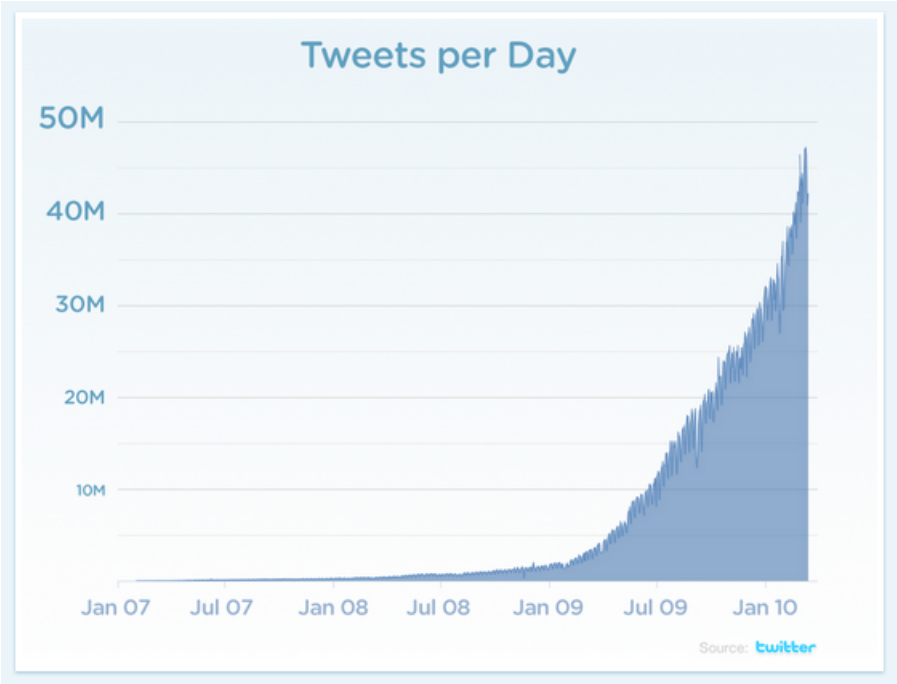


Figure 2.2.: Tweets per day [2]

is around 180% [16]. Tweets per day is just one thing that motivates us to use this social network in times of outages. Another reason is mainly represented by his versatility.

1	AH1N1 - Swine Flu
2	Mubarak - former Egyptian President
3	Easter - Christian holiday
4	Cairo - capital of Egypt
5	#prayforjapan - sentiment following the March earthquake and tsunami
6	Chernobyl - site of nuclear disaster in 1986
7	Libia/Libya - site of an ongoing civil war
8	Fukushima - Japanese nuclear power plant

Table 2.1.: Trending Topics first half 2011

Twitter essentially is a combination of various forms of communication and its primary difference is that tweets, are limited to 140 characters or less. Although the tweets are really short, they changed significantly from normal messages and are evolving toward links to content on the web, discussions about important topics using hashtags (keyword marked with #) and most importantly real time news for people in a natural disaster or political uprisings. Moreover relationships are not reciprocal, and they are not based on trust but on authority, since it is used from famous people or organizations in order to spread news. Twitter therefore is not like Facebook and all its features are really well suited to spread information rapidly, especially during emergencies.

2.1.1. Natural Disasters

Due to its increasing popularity, Twitter has been used heavily by affected people during natural disasters [17] in order to organize the emergency response as well as to find family members and friends. Short messages in fact can be easily written and spread among a very large number of people. Adam Ostrow, an Editor in Chief at Mashable [1], a social media news blog, wrote about the interesting phenomenon of real-time media as follows :

> "Earthquakes are one thing you can bet on being covered on Twitter first, because, quite frankly, if the ground is shaking, you're going to tweet about it before it even registers with the USGS and long before it gets reported by the media."

The most recent example is represented by the Japanese earthquake in spring 2011. Unfortunately, during such situations the network infrastructure might fail, showing its weakness in times of the greatest need. In the hardest hit areas of a disaster therefore connectivity is really poor at best and access services like Twitter might be problematic. Table 2.1 shows the number of inoperative base stations in Japan on March 12:

Operator	# inoperative
NTT DoCoMo	6720
KDDI	3800
Softbank	3786

Table 2.2.: Number of inoperative base stations after the earthquake in Japan [10]

In addition to the devastating impact of the earthquakes and other natural disasters generally network overload occurs often since a huge number of people attempt to call or send a message to contact loved ones. During the Japanese disaster computer-based Internet traffic and especially mobile one increased in volume for two days following the disaster [18].

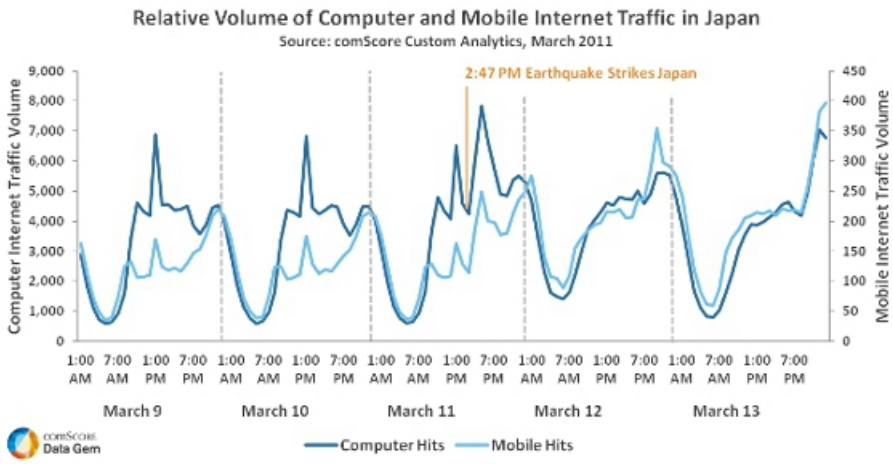


Figure 2.3.: Volume of Internet Traffic in Japan [3]

[1] http://mashable.com

This does not mean obviously that the mobile infrastructure is more reliable and it can be seen just as a sign of the mobile media importance. The Japanese communication providers were all affected in fact by multiple outages [19] and most of the information flow came generally from outside the heavy hit area. The majority of the traffic was via Twitter and Facebook that played an important role to share information and organizing donation and first aid activities.
Besides this last tragic event, Twitter was used also to provide situation updates during the 2007 California wildfires [20], the 2009 Red River Floods [21], and the 2009 Oklahoma Grassfires [21] not to mention the role of major source of information after the earthquake in Haiti[22]. Catastrophic natural disasters like these clearly highlight the need for some kind of alternative, distributed networks that are missing today for the general public.

2.1.2. Political Uprisings

Recently Twitter along with other social network and Youtube have played an important role during political uprisings. An important case was the recent Egyptian "Revolution" during January 2011, where Twitter has been used as a vital source of information and for organization purposes. The government understood the importance of social media and communications, therefore managed to shut down almost completely Internet leaving millions of Egyptians without access to Internet. The shutdown involved the withdrawal of 2576 networks from BGP routing tables. Only one ISP out of 10, Noor Data Networks, was still working properly. It connects Egypt to the outside world via an undersea cable operated by Telecom Italia. According to this data hence 88% of the "Egyptian Internet" was removed from the global network. Figure 2.4 shows the situation.

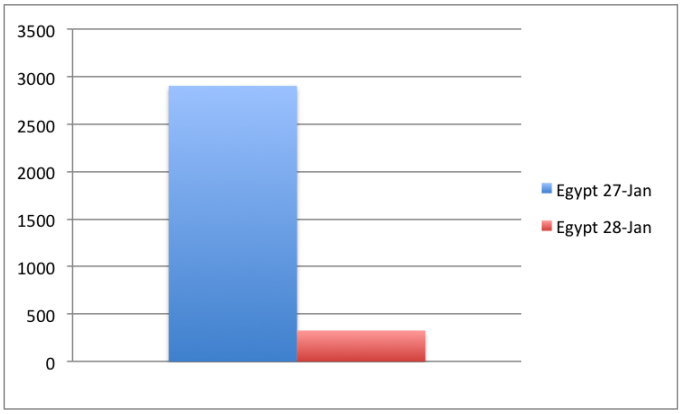


Figure 2.4.: Egyptian Routes [4]

Egyptian authorities are also reported to have blocked Internet access by shutting down official Domain Name Servers (DNS) in Egypt. Google and Twitter declared themselves willing to help the Egyptian public. They teamed up and launched a new speech-to-text system that converted voice messages into tweets that were published online with the "#Egypt" hashtag[23]. This recent case therefore points out again the need of solutions in order to fight against censorship of dictatorship regimes. It is not just limited to the Egyptian revolution since in places such Iran, Bahrain and China Internet access can be limited, content filtered and freedom of speech is heavily restricted.

2.2. Android OS

Google's Andy Rubin describes Android as [2]:

> "The first truly open and comprehensive platform for mobile devices. It includes an operating system, user-interface and applications, all of the software to run a mobile phone, but without the proprietary obstacles that have hindered mobile innovation"

In simple words Android is a software stack that includes lots of components, not only the most famous ones like the operative system and the user interface. It is based on the Linux kernel that provides the necessary low level interface with the hardware and it is released as a free, open-source software for mobile devices.

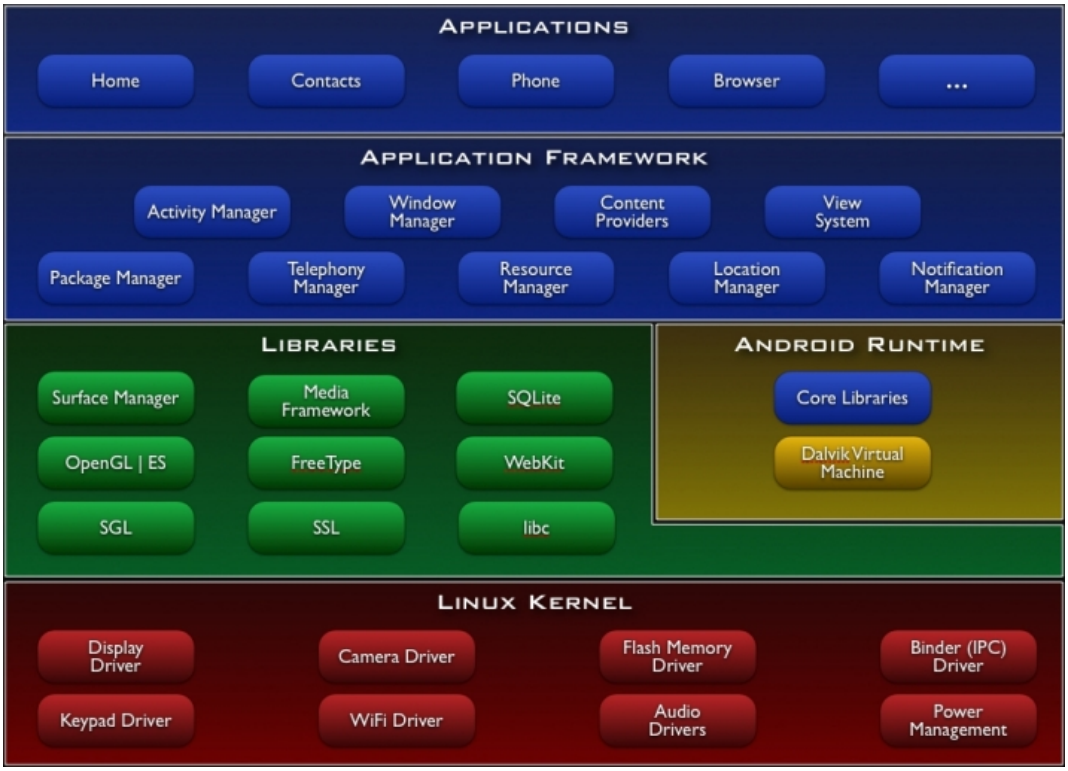


Figure 2.5.: Android Architecture [5]

What is really attractive is its open philosophy, which gives the possibility to everyone to fix or replace components and develop applications that look and function as a developer wants. Android phones are sold with a set of standard pre-installed applications, but, as said before, they can be replaced without limitations. All these characteristics make Android really appealing for device manufacturers and probably this is the key of its success.

[2] http://googleblog.blogspot.com/2007/11/wheres-my-gphone.html

2.2.1. Development on Android

Android provides an open development platform, therefore developers can design really innovative and full of features applications. Designing applications for Android is extremely easy compared to other platforms due to the powerful SDK, no fees, rich documentation and online support by open source communities.

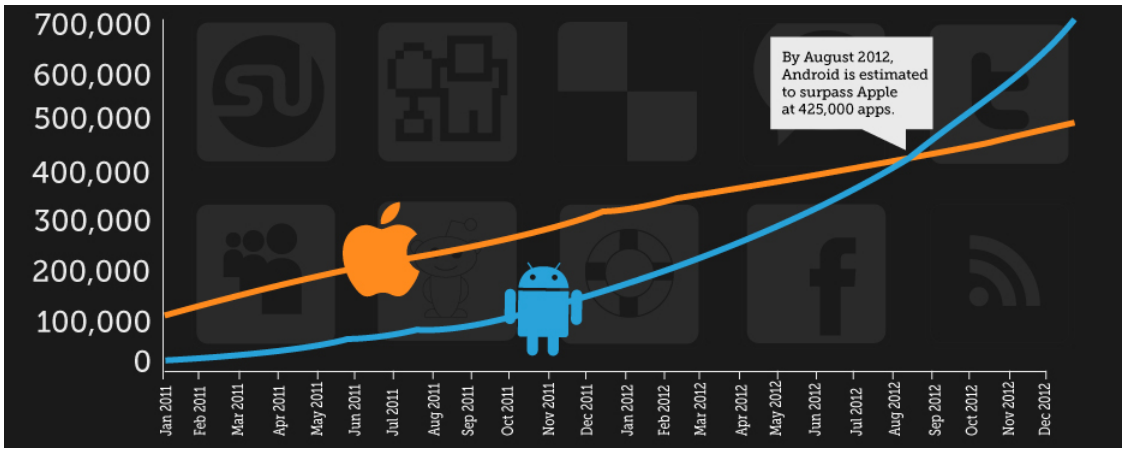


Figure 2.6.: Application development[6]

All Android applications are written using the Java language and writing one does not require any developer certification. Even people who have never programmed for mobile devices before can create nice applications. Once an application is ready it can be published easily on the Android Market for distribution. Moreover, the published applications do not undergo any approval process, hence are available on the market straightaway. All the reasons explained so far were a great incentive for choosing it as the platform to be used to design the application.

2.2.2. Market Share

Last but not the least important reason, which pushed to develop the disaster enabled Twitter client on Android was clearly the market share and his rapid growth in the past two years. Up to date (summer 2011) Android has around 38% of market share and it is predicted to reach 50% after 2012 as shown in figure 5.3.

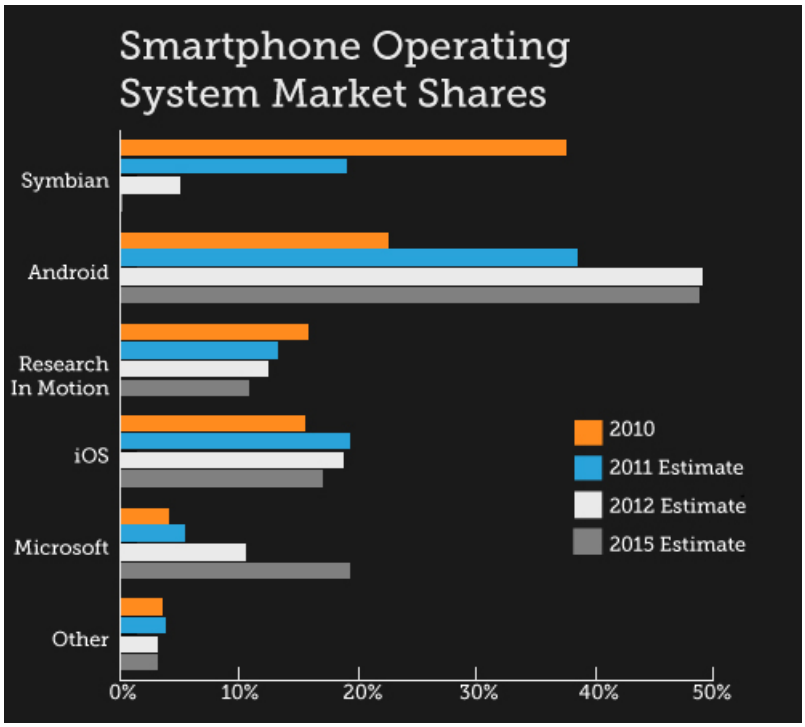


Figure 2.7.: Android Market Share [6]

Hence, this was another good incentive to choose Android as a platform for building the Opportunistic version of Twitter.

3. Related Work

This chapter shows some related work. First Delay Tolerant Networks are explained since the main purpose of this thesis was to allow communication exploiting infrastructure-less networks among mobile devices carried by people. Then some previous opportunistic projects are discussed briefly, namely Haggle, Mobiclique, D-Book and DT-Talkie.

3.1. Delay Tolerant Networks

Traditional networks both wired and wireless assume that an end to end path between nodes always exist. Partitions are usually considered failures and in case an alternative path can not be found, the delivery of data fails. However for some wireless applications such as environment monitoring, deep space communications and Vehicular networks this assumption does not hold anymore. This might be due wireless propagation effects, mobility or energy saving techniques implemented by mobile nodes. In these cases, devices might not be in communication range most of the time, however, the possibility to communicate is still desirable. Such kind of environments that are characterized by intermittent connectivity are usually called Delay Tolerant Networks (DTNs) [24][25] .
Basically, they aim to create infrastructure-less networks among mobile devices carried by people. Network partitions are not seen anymore as failures and they are connected exploiting mobility. Although they are not suited for real time audio or video, they can be useful in a lot of scenarios where some delay can be tolerated. Moreover, they are becoming more and more important for challenged environments such as natural disasters areas or for circumventing censorship of dictatorial governments. A number of routing techniques have been proposed for this kind of scenarios. They are mainly based on the store-carry-forward paradigm. Once a message is received by a node, it is stored in the local memory and carried around while the device is moving around. Once a forwarding opportunity appears it sent to other nodes in communication range.

3.2. PodNet

Podnet is a project of the Communication Systems Group at the ETH Zürich together with the KTH Stockholm [7] [26]. Basically, it extends the traditional podcasting concept to delay tolerant networks. Podcasts are fetched by a Podnet server and spreaded in the wireless area network using fixed gateway as well as the mobile nodes. This is feasible thank to actual smartphones capabilities. The users of Podnet can subscribe to a podcast, or even create their own one, and distribute it to the other mobile devices in communication range. Podnet therefore allows mobile devices without Internet connectivity to access information and provides an ad hoc domain where they can act as source of information as well. Figure 3.1 shows the architecture of PodNet.

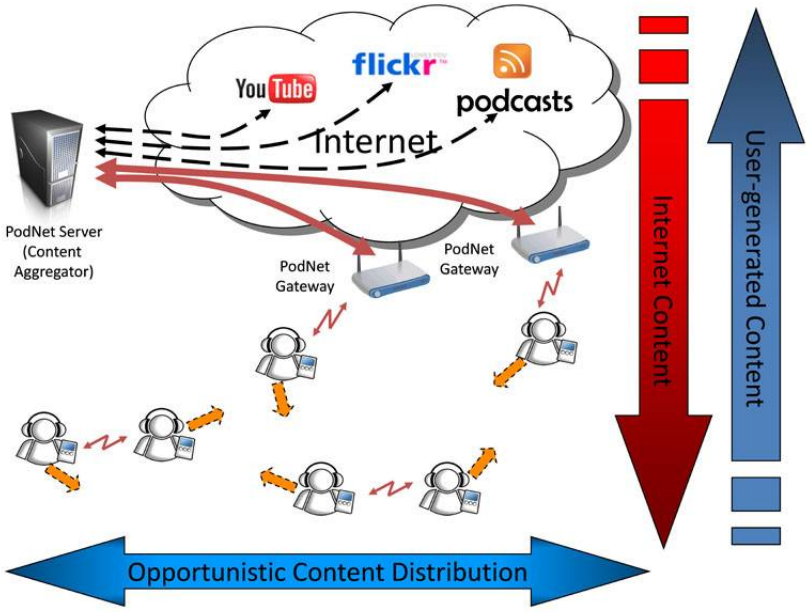


Figure 3.1.: Podnet Architecture [7]

3.3. Haggle

Haggle is an open source multi platform network architecture for delay tolerant networks[1][27]. It exploits opportunistic contacts among devices to forward data toward a destination. It uses a search based network architecture and provides to applications a completely new architecture for addressing, data storage, devices discovery and resource management. The core of the Haggle architecture is the abstraction layer that hides the details of data transmission and provides spacial and temporal decoupling using a data-centric communication model with publish-subscribe APIs. In order to solve the problem of mapping between users and data uses a search based solution. Through all this mechanisms Haggle allows to separate the application logic from the underlying networking technology. Hence an email client or a photo sharing application [28] running on a mobile device would delegate the Haggle system to handle the communications. It will choose dynamically the best available connectivity and protocol for the situation.

3.4. Mobiclique

Mobiclique is a mobile social software for the Windows Mobile Platform based on the Haggle network architecture. It allows users to extend the existing infrastructure based online social networks to the opportunistic encounters in real life. Therefore it does not rely on network infrastructure for connectivity, but on the ad hoc capabilities of smartphones. Mobiclique performs the bootstrap of user information from the central servers and then enables connectivity whenever to peers are in communication range in the physical world. When people meet opportunistically updates are exchanged in case they have some kind of connection in the virtual world. The communication technologies used are Bluetooth and WiFi. The data exchange is performed according to the

[1]The Android version requires rooted phones

store-carry-and forward paradigm and can be unicast or multicast. Unicast messages are generally forwarded over the friendship graph whilst multicast is used for instance to forward information to interest groups.

3.5. D-Book

D-Book was developed as part of a master thesis at the Georgia Institute of Technology [29]. It is an implementation of a fully distributed version of Facebook. It applies the concepts of the DTNs to a famous web based social networking application. It is based on the DTN# a .Net-based implementation of the DTN bundle specification [30].

3.6. DT-Talkie

DT-Talkie is an application that allows walkie-talkie communications over delay tolerant networks. This kind of service was traditionally based on infrastructure, both cellular and operator independent wireless. With DT-Talkie it is not necessary anymore since nodes record audio messages that after being encapsulated in the suitable data structures are transmitted via the store-carry-and-forward paradigm to other phones. The bundle protocol agent used for this application is the DTN2 developed by the DTNRG [24]. DT-Talkie was implemented for Nokia Internet Tablets. It has been ported also for Linux and Mac notebooks.

3.7. Discussion

For this master thesis none of the previous solutions have been used. Therefore, the application was developed from scratch without relying on existing platforms. The main reason for this choice was essentially the fact that some of them were too specific (PodNet, DT-Talkie) or too generic (Haggle). Basically what we needed was just an suitable existing social network to be extended with opportunistic features. This did not need any advanced opportunistic stacks (Haggle) or audio messages. Moreover some of the previous projects were not specifically design for Android phones or they require rooted phones, which is not a good idea for applications to be publish on the Android market.

4. Applications Survey

Besides the official Twitter client there are some other ones from companies as well as some open source ones. This chapter will survey the most important application and explains the reasons for the final choice to develop a new one from scratch.

4.1. Commercial Apps

This section describes briefly some famous commercial Twitter clients. Almost all of them can be downloaded from the Android Market for free, however, if one wants to use more advanced features should pay something between 2 and 5 US Dollars.

4.1.1. Reading Tweets

	Twitter	HootSuite	Seesmic	Swift	Touiteur	TweetCaster	Twicca	Twidroid
Load timeline at launch	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Remember timeline position	No	Yes	Yes	No	Yes	Yes	Yes	Yes
View geotagged tweets on map	Yes	No	Yes	No	No	No	Yes	Yes
View nearby tweets	Yes	No	No	No	No	Yes	No	Yes
View nearby tweets on map	No	No	No	No	No	No	No	No
View lists	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes
Enlarge profile pictures	No	No	Yes	No	Yes	Yes	No	Yes
View conversation threads	No	Yes	No	Yes	Yes	Yes	Yes	Yes
Change font size	No	No	Yes	No	Yes	Yes	Yes	Yes
Picture thumbnails in timelines	Yes	No	No	No	No	No	No	No
	Twitter	HootSuite	Seesmic	Swift	Touiteur	TweetCaster	Twicca	Twidroid

Figure 4.1.: Functionalities reading tweets [8]

As figure 4.1 shows, Twidroid is clearly the most complete client for browsing tweets, closely followed by Twicca and TweetCaster. Surprisingly the official Twitter client lacks some nice features like loading timeline at launch, changing font size and view conversation threads.

4.1.2. Sending

	Twitter	HootSuite	Seesmic	Swift	Touiteur	TweetCaster	Twicca	Twidroid
Insert location in tweet	Yes	No	Yes	No	No	Yes	Yes	Yes
Old / new retweet options	New	Old	Both	Old	Both	Both	Both	Both
Reply to all mentions	No	Yes	Yes	No	Yes	Yes	Yes	Yes
Look up user names while composing	Yes	No	No	No	Yes	Yes	Yes	Yes
Upload photos / videos	Photos	Photos	Both	Both	Photos	Photos	Both	Both
Photo upload service options	2	1	4	4	2	2	1	5
Video upload service options	No	No	3	2	No	No	1	Premium
URL-shortening service options	2	1	3	1	Premium	2	1	3
	Twitter	HootSuite	Seesmic	Swift	Touiteur	TweetCaster	Twicca	Twidroid

Figure 4.2.: Functionalities sending [8]

As shown in Figure 4.2, Twidroid appears again the most complete in terms of features composing as well. Twicca follows closely cause the restricted URL-shortening. Seesmic is characterized by a good amount of features as well and it is the only one that can upload photos and videos for free. It is just missing the look up of usernames while composing.

4.1.3. Wrap up

Almost all the applications look pretty valid but Twidroid can be easily identified as the best one in terms of functionalities and user experience even in the basic version. However since it is not the official Twitter application it cannot benefit from the integration with the mobile OS. It means that some functionalities provided b the Android system can not be exploited such as posting a picture from the gallery.

4.2. Open Source Apps

When evaluating Twitter open source clients the number of candidates is much lower as one might expect. Twitter announced during 2010 of open sourcing the code but up to date (July 2011) nothing has been released.

	AndTweet	Twitli	C:tweet	TwitterDroid	NanoTweeter	Twift	Twimight
Code available	Yes	Yes	Yes	Yes	Yes	Yes	Yes
APIs	No	No	No	Yes	No	Yes	Yes
Runs	Yes	Yes	Yes	Yes	Yes	Yes	Yes
OAuth	Yes, Optional	No	Not Browsable	No	No	No	Yes
Features	Limited	Excellent	Basic	Limited	None (notify)	Limited	Limited + Disaster
Updated	Often	No	No	No	2010	2010	Yes
Code Comments	Yes	Some	Some	Yes	Some	Japanese	Yes
Code Complexity	High	Extreme	Low	Low	Low	Medium	Medium
User Interface	Simple	Good	Good	n.a.	Almost none	n.a.	Good

Figure 4.3.: Open Source apps comparison

Figure 4.3 shows the comparison between the most important clients. The last column on the right hand side represents the application developed for this master thesis. The table clearly shows that all the Open Source clients with the code available in a online repository have one or more fields marked with red.
The most serious problems that affects all the surveyed applications are lack of APIs and/or support for OAuth authentication. The first problem affects heavily the code complexity since it would make necessary access low level functions for retrieving or posting the information online. The latter even prevents the login therefore would be impossible perform any kind of operation.

4.3. Final Choice

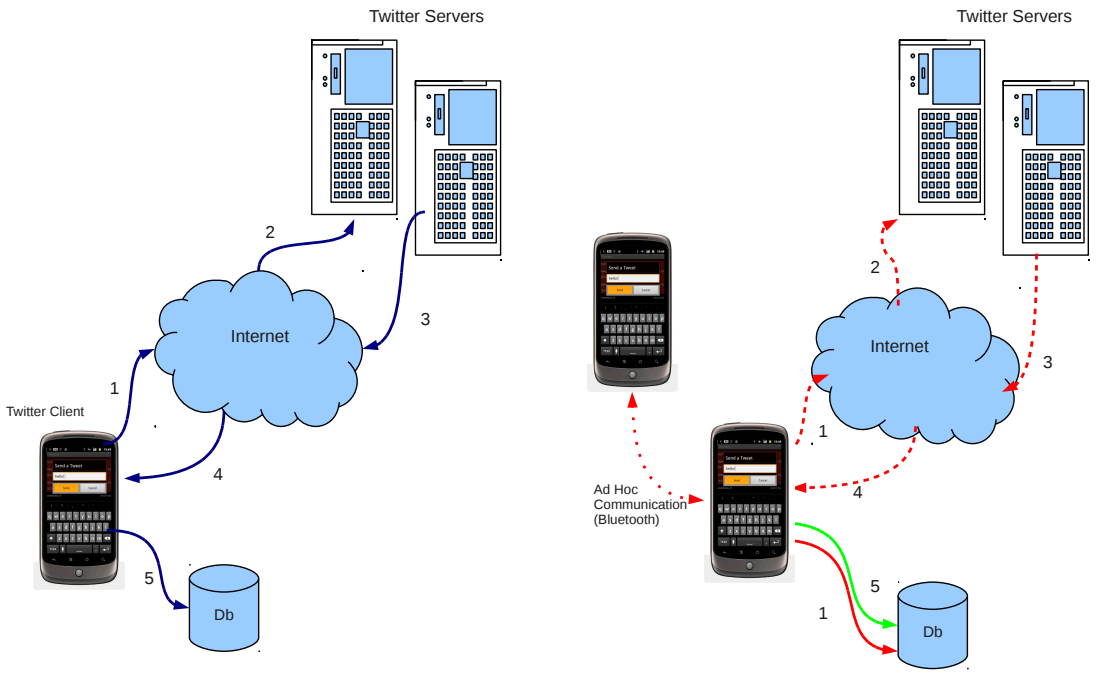
After evaluating carefully the open source apps available, the final choice has been to develop a basic disaster enabled Twitter client from scratch starting from the Marakana's online tutorial [31] and the BLOA OAuth example [32]. The Bluetooth communications are based on some code available on the Android developers web page [5]. Certainly, it has been really useful in order to understand better programming on Android OS.

5. Application Architecture

The purpose of this chapter is to give a good insight of the design decisions and the architecture of Twimight. It is necessary to understand how the client works during normal, and particularly disaster mode, where opportunistic communication is used. Section 5.1 describes generally how the application works during both modes. Then section 5.2 focuses on the communication with the Twitter servers. Section 5.3 describes how the application works both in normal and disaster mode separately showing the respective diagrams. The GUI and the main functionalities offered during disaster mode are explained in section 5.4. Data Structures, including packets and the Database architecture are shown in section 5.5. Finally, section 5.6 explains the reasons behind using the Bluetooth standard, the neighbor discovery mechanism and the message flow once a connection has been established.

5.1. Application Description

As stated in section 3.7, instead of relying on existing opportunistic stacks, we decided to develop the application from scratch starting from a specific application (Twitter), and a platform (Android). Hence generic stacks like Haggle would be too complex and cumbersome, so it was better to start with a simpler approach, better suited for the purpose. Therefore, this project approaches the problem from a different perspective. Instead of developing an opportunistic network first, and then deploying applications to use it, we started from a famous application and then we extended it with an additional feature to allow opportunistic communications in case of breakdown of the network infrastructure. Since many people is already used to mobile versions of Twitter, we want to minimize the additional confusion for people in a distress situation. The client works exactly in the same way as a normal one even when the disaster mode is active.
Once it is enabled (either manually or automatically) it switches Bluetooth on and it starts the disaster operations in the background without any additional burden for the end user. All the features will be described in detail throughout this chapter.



(a) Normal mode

(b) Disaster mode

Figure 5.1.: Normal and Disaster mode.
Continuous blue lines mean operations that are performed with success. Dashed red lines indicate an attempt in disaster mode while a continuous red is used for actions without problems in disaster mode.

Figure 5.1 shows how the application works, both in normal and disaster mode. During normal operations, after successful OAuth authentication, the client publishes and receives tweets from central servers using the Twitter APIs. The most important information is saved in a local SQLite Database, which is suitable for mobile devices [33]. During this mode of operation, it does not differ too much from the existing clients after all. Once the disaster mode is activated by the user (see Fig. 5.6c), the clients switches to a "hybrid" mode in which opportunistic communication is enabled. That is because if the client detects Internet connectivity, it will try to use it as well to speed up the spreading of information, especially in case the client is carrying data for other users (i.e. tweets received opportunistically). The opportunistic communication is achieved using the Bluetooth radio embedded in almost all the Android phones. The disaster tweets are stored in the local database as well, but in different tables both for efficiency and security reasons. After that, every time two devices are in communication range relevant tweets are exchanged. Details are given in the following subsections.

5.2. Communication with Twitter

5.2.1. Twitter APIs

Twitter allows third party applications to access user data via its APIs[1]. To develop the Twitter client, the open source JTwitter APIs have been used [34]. Essentially, they are a small library that provides access to the low level Twitter http based API methods. They provide methods to update a status, read a timeline or send a direct message to a friend. The most important methods used for the application are included in the class "Twitter".

5.2.2. OAuth Authentication

Since the end of August 2010, third-party Twitter clients are allowed to communicate with the central servers only using the secure OAuth method [35]. The idea of OAuth is prevent malicious apps from stealing the credentials of a user since neither username nor password are stored on the clients. Without using OAuth would be like going to a restaurant and giving the credit card and the secret code to the waitress. This kind of problem has become more and more critical since a lot of services and web pages are distributed ones and can be accessed from mobile devices. OAuth comes to help since it gives the possibility to grant third-party web pages or mobile clients limited access to service providers. For instance a Twitter user can grant a mobile client access to his private profile without sharing any private data with it.

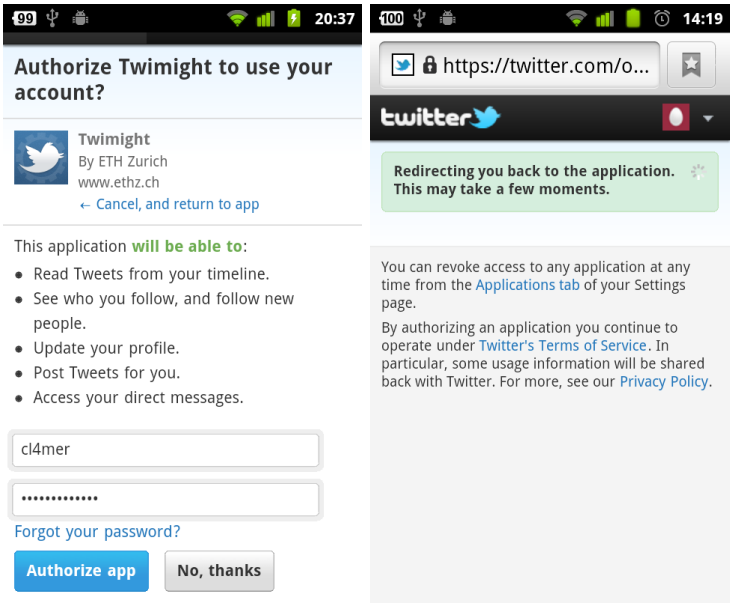


Figure 5.2.: Twimight Authorization

Additionally, an OAuth enabled client can continue working even after changing the Twitter password on the Twitter web site. Access to an application can be easily revoked just by logging into Twitter and changing the settings. Figure 5.2 shows the Authorization for Twimight. The user authenticates directly on Twitter which issues the mobile client specific tokens. Every third-party Twitter application like Twimight has to be registered first on the central servers and will be identified by a consumer key and secret. Figure 5.3 shows the registration process for Twimight.

[1] https://dev.twitter.com/docs

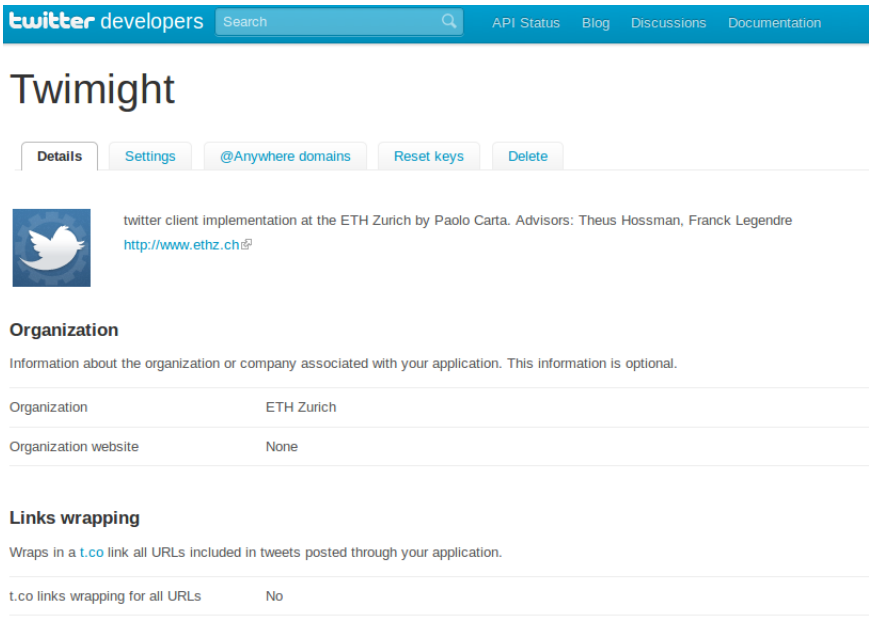


Figure 5.3.: Twimight Registration

5.3. Application Diagrams

After a general introduction of the application it is necessary to give more implementation details.

5.3.1. Normal mode

Figure 5.4 shows the Twimight diagram including the most important functionalities of the client. After successfully authenticating, the client stores the OAuth token and secret in a shared preferences file. This is a lightweight mechanism Android provides to store data with good privacy level, since only Twimight will be allowed to access that file. The secret tokens can be used for every successive login to the Twitter servers using the Twitter APIs. Upon login, the app checks whether it has to generate a new key pair for encryption purposes (For more details see Security chapter) and starts the Updater Service whose purpose is download all the data for the authenticating user (i.e. timeline, mentions, favorite tweets and direct messages). Besides starting after the first login, the Updater Service is started also when Android boots. The service runs in a separate Thread, since network operation might take time to complete, and would otherwise block the main user interface.

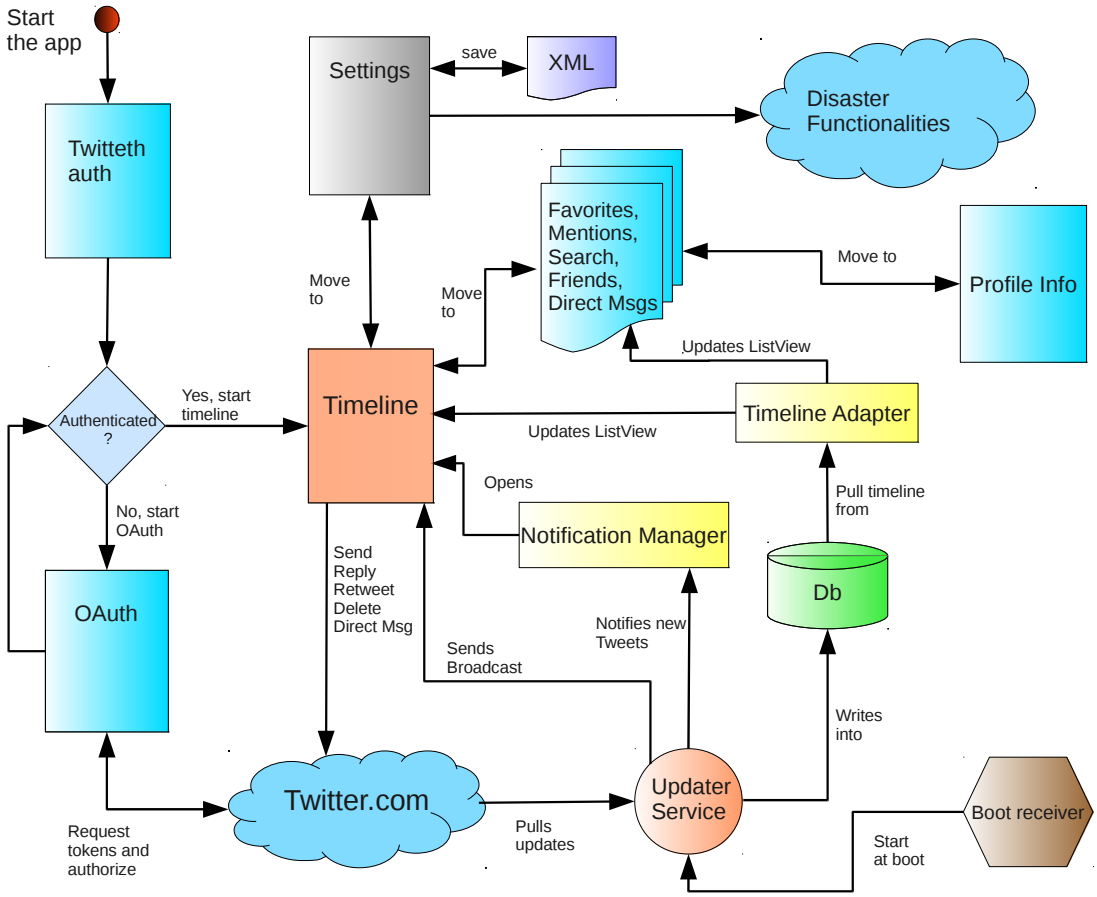


Figure 5.4.: Twimight Diagram in normal mode

The data downloaded by this background service is written into the appropriate tables in the SQLite database (See section 5.5.2) thank to a database helper class. Whenever new data is available, the user is notified using the Notification Manager. All the data in the database can be visualized via a custom adapter, that acts as an interface between the GUI and the underlying database. The most important activity in Twimight is clearly the Timeline, from which we can access all the other views and the settings from which we can enable the Disaster Mode. For more details regarding the menus and dialogs available see section 5.4.

5.3.2. Disaster mode

After explaining the basic behavior of Twimight, it is necessary to dig deeper on the most attractive and innovative feature of this client, the disaster mode. Figure 5.5 expands the cloud that represented the disaster functionalities in the previous figure. In order to make the diagram as clear as possible, only the most important classes and/or functionalities are shown.

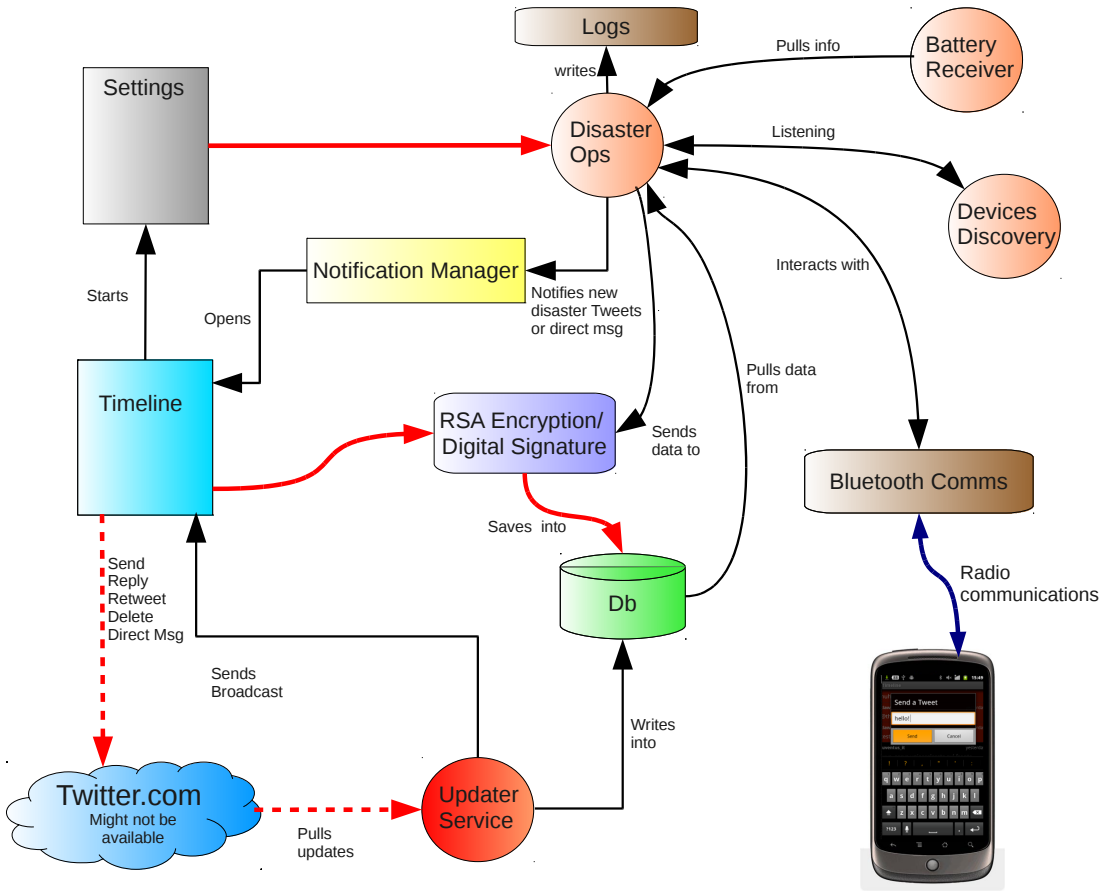


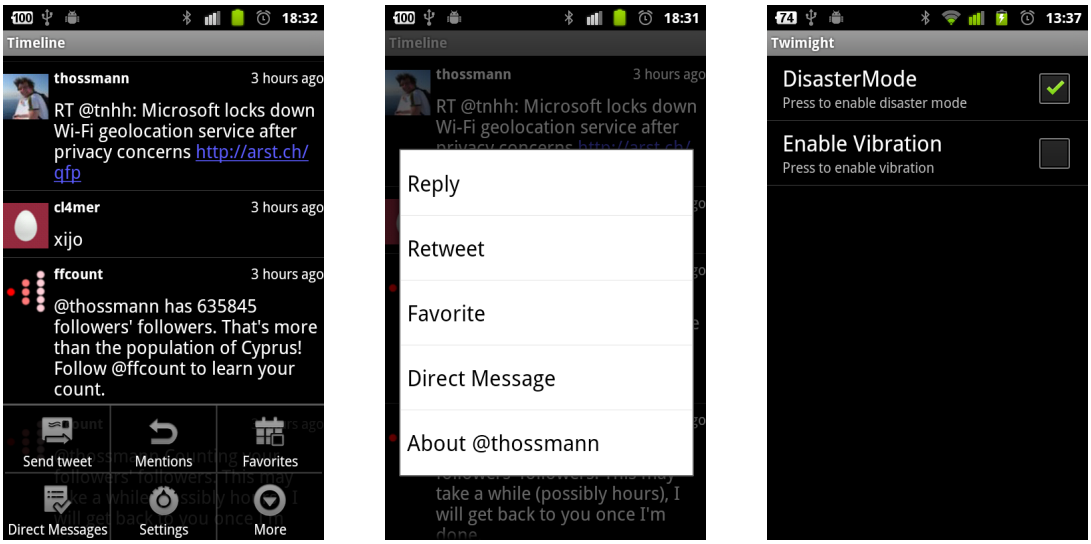
Figure 5.5.: Twimight diagram with disaster mode
Continuous black lines mean operations that are performed with success. Dashed red lines indicate attempts in disaster mode while a continuous red is used for user actions without problems in disaster mode.

Once the disaster mode has been enabled from the settings, the application immediately switches Bluetooth on and starts some background processes: the Disaster Operations, Devices Discovery, Battery Receiver, the Airplane mode listener and the Bluetooth Communication Manager. Moreover, in order to maintain the CPU awake even in case the display is switched off, the application automatically acquires a special wake lock object from the operating system. Twimight starts then periodically scanning for devices around. Once a peer has been discovered it attempts to connect to it (after pairing, in case it is required). After connecting, devices exchange relevant information, thereby spreading them epidemically. The received tweets (after processing and signature verification), are stored in the appropriate tables in the local database and the user is notified by vibration and flashing lights. All the tweets and direct messages exchanged are marked as disaster

information. This is important for the correct visualization in the main GUI through the custom adapter.
During disaster operation, the client checks periodically if network infrastructure is available, even though it might be just for a few minutes. In case connectivity is detected, Twimight starts publishing all the relevant information on the Twitter servers. In the current version every client publishes the tweets the user created as well as standard and local disaster retweets. The first type of retweet is published as usual whilst the disaster retweet is published as an old style ones. An old style retweet is simply identified by the RT @username tag whilst a new one is linked to the original tweet using its unique Id. This design choice has been made in light of redundancy and efficiency reasons. If every client published everything this would lead to an exponential number of tweets online. For instance assuming that there are 100 users in disaster mode and each one sends 10 Tweets to all other 100 users. This means that each user carries 1000 tweets and once normal operation is resumed we would have a storm of 100*1000 retweets.

5.4. Graphic User Interface

This section will explain in detail the Graphic User Interface of Twimight, mainly focusing on the disaster functionalities. As every Android application, Twimight is characterized by multiple Activities and background Services. Every activity for an Android application must be declared in the manifest file in order to be visible for the system. See appendix B and C for more information. Every activity represents a single screen with a User Interface (UI) on which the user can perform different actions. Every activity extends the "Activity" class and implements the most important callback methods that are called by the OS when specific events occur. The UI of an activity consists of a hierarchy of views. It can be defined with an XML layout file or programmatically in the class.

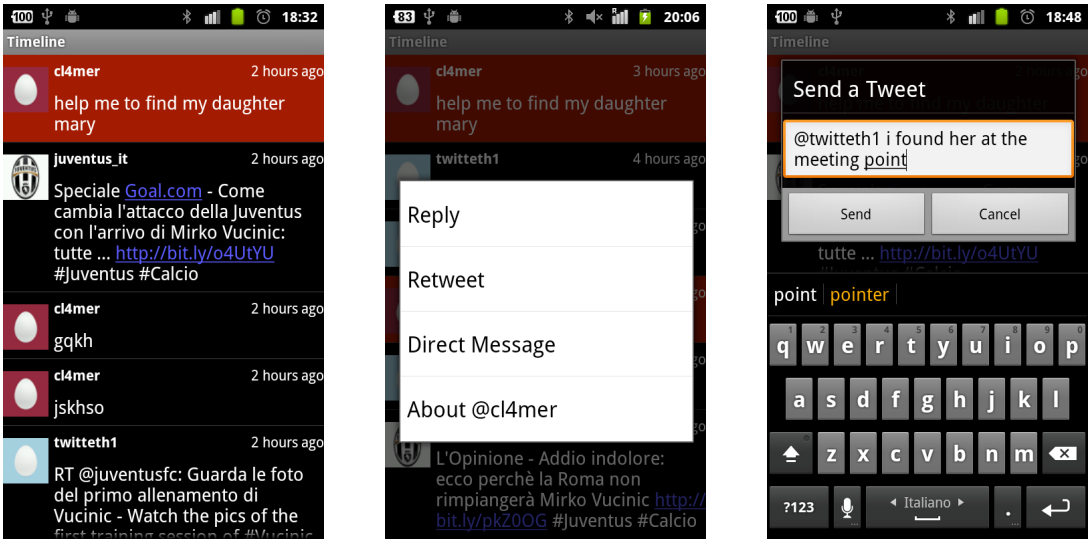


(a) Normal timeline and menu (b) Normal context menu (c) Settings

Figure 5.6.: Twimight screen-shots

The main activity for Twimight is the Timeline. Its UI is defined with an XML file. In the timeline the user can read the tweets, open the menu to start other activities or open a dynamic

context menu with a long click on a tweet. Figure 5.6 shows some screen-shots of Twimight during normal operation. Once the user activates the disaster mode in the settings menu, there are a few changes in the user interface and in the background processes that are meant to improve the user experience in a distress situation. The first difference can be observed sending a tweet. If Twimight detects no connectivity the tweet is stored locally as usual but it is visualized in the main timeline along the other tweets with a red background as shown in figure 5.7a.

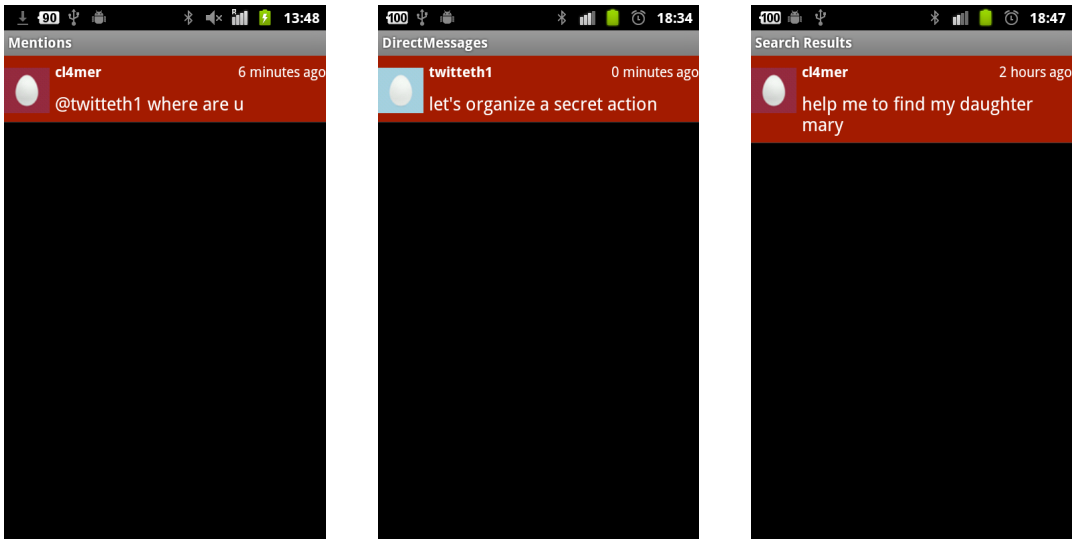


(a) Disaster tweet marked (b) Disaster context menu (c) Disaster reply

Figure 5.7.: Disaster Mode screen-shots

The context menu that appears with a long press on a disaster tweet (similar to a right click in a computer) contains 4 options: Reply, Retweet, Direct Message and About (Figure 5.7b). The reply function allows a person to send a tweet back to a user automatically including the @username tag.

The retweet works a bit differently in disaster mode. Since the spreading of tweets is done epidemically (details in section 5.6.3) the reason for retweeting in disaster mode is mainly related to publishing the disaster tweets once connectivity is detected. A disaster retweet in fact will be published on the central servers along with the user generated ones. If a disaster message received epidemically is not marked as a "disaster retweet" it will not published online, as explained in the previous sections. Therefore the main purpose of the retweet in this case is a human-based filtering system. Only local disaster tweets that are retweeted will be published online. During disaster mode, is possible to send a direct message to a specific user as well. It can be sent from different activities and it is spread epidemically with all the other relevant information. However, since by definition it might contain private information, it is encrypted with a public key scheme, so that only the receiver can read it. The other devices just act as mules to relay it to the correct destination. Figure 5.8b shows a received disaster Direct Message. More details about encryption are given in chapter 7



(a) disaster mentions (b) Disaster direct message (c) Disaster search results

Figure 5.8.: Disaster Mode screen-shots 2

The other important functionalities in disaster mode are Mention tweets and the Disaster Search. Basically, when in disaster mode the user opens the mentions activity, a background thread scans the disaster table in order to find tweets in which the user has been mentioned. In case it finds some, they are shown with the classical red background. Assuming that tweets are spread epidemically, they will be received by all peers. One might be interested just in a specific user or hashtag and might want to display only related tweets. This is possible in disaster mode since the user can perform a search for specific data on the local database.

5.5. Data Structures

This chapter explains the data structures used for Twimight, namely: the packets exchanged opportunistically and the local SQLite Database.

5.5.1. Packets

First the basic data structures to store a simple tweet or direct message are explained, then the actual packets types are discussed. Basically, there are two different types, namely Hello and Data packet.

5.5.1.1. Signed Tweet

The signed tweet is basically the class that contains all the relevant information for a tweet. It is added into the list of tweets carried by the data packet. It is represented in figure 5.9

Id	Timestamp	User Id	Text	User	Retweet	Sent	Hops	Public key	Signature

Figure 5.9.: Signed tweet structure

The relevant fields are:

- Id: hashcode of a message that uniquely identifies it in the opportunistic network.
- Timestamp: represents the time instant when a tweet was created.
- User Id: unique code that identifies a user.
- Text: the actual message.
- User: nickname of the sender.
- Retweet: binary field which specifies if it is a tweet or a retweet
- Sent: other binary field that is used to communicate if the tweet was published or not online.
- Hops: the number of hops for the tweet in the opportunistic network. It can be used as a TTL.
- Public key: signed public key of the sender for digital signature verification.
- Signature: digital signature used to verify authenticity of the tweet.

5.5.1.2. Direct Message

The direct message is the class that contains all the relevant information for it sent opportunistically. It is represented in figure 5.10

Id	Timestamp	User Id	*Text*	*Sender*	Recipient	Sent

Figure 5.10.: Direct Message structure

The relevant fields are:

- Id:uniquely identifies a direct message in the opportunistic network.
- Timestamp: represents the time instant when it was created.
- User Id: unique code that identifies a user.
- Text: the actual message, encrypted with the receiver's public key.
- Sender: nickname of the sender encrypted with the receiver's public key.
- Recipient: nickname of the recipient.
- Sent: binary field that is used to communicate if the direct message was published or not online.

5.5.1.3. Hello Packet

The hello packet is the first one that is exchanged when two peers are in communication range. It is mainly used to optimize the information spreading in the opportunistic network.

Type	Close Connection	Timestamp	# Messages

Figure 5.11.: Hello packet structure

The fields shown in figure 5.11 are:

- Type: used to identify the type of packet (type 1 or 2).
- Close Connection: binary field used to ask to close the connection.
- Timestamp: used for synchronization purposes and to identify the last tweet received (see section 5.6.3).
- Number of messages: it is used in order to inform the other peer about the number of messages seen in the previous encounter, if any (see section 5.6.3).

5.5.1.4. Data Packet

The data packet contains the relevant tweets and direct messages to be sent to a peer.

Type	Close Connection	Timestamp	Tweets	Direct Messages	# Tweets

Figure 5.12.: Data Packet Structure

The fields shown in figure 5.12 are:

- Type: used to identify the type of packet (type 3).
- Close Connection: binary field used to ask to close the connection.
- Timestamp: used for synchronization purposes and computing statistics (such as delivery delay).
- Tweets: contains a list of tweets each one formatted as explained in subsection 5.5.1.1 .
- Direct Messages: contains a list of encrypted direct messages each one formatted as explained in section 5.5.1.2 .
- Number of Tweets: used to communicate to other peers the number of tweets currently stored in the database.

5.5.2. SQLite Database

Behind the GUI that dynamically interacts with the user in real time, all the data are stored in a local SQLite Database in the backend. It is an open source database supported by Android, that does not require any setup and it is well suited for mobile devices, since it occupies little memory at runtime. It represented therefore a lightweight solution with good security features. Creation and update is managed by a helper class implemented in Twimight. Then, data can be written and read from it by to an interface class.

5.5.2.1. Tables

The Twimight database contains 10 tables: Timeline, Disaster, Addresses, Favorites, Mentions, Profile Pictures, Direct Messages, Search Results, Peers, Direct outgoing messages (for the disaster mode). In order to reduce redundancy, all the tables respect the third normal form (i.e. there are no fields which depend on other attributes that are not the primary key). They are shown in detail in figure 5.13. Every table has a Primary key that is written in bold and not null attributes are highlighted with a black dot.

Timeline

Id	**INTEGER**
created_at	INTEGER
status	TEXT
userCode	INTEGER
isDisaster	INTEGER
isFavorite	INTEGER

Favorites

Id	**INTEGER**
created_at	INTEGER
status	TEXT
userCode	INTEGER

Addresses

mac	**TEXT**
met_at	INTEGER
tweetsNumber	INTEGER

Profile Pictures

user	**TEXT**
image	BLOB
created_at	INTEGER

Disaster

Id	**INTEGER**
created_at	INTEGER
added_at	INTEGER
status	TEXT
userCode	INTEGER
sent_by	TEXT
isFromServer	INTEGER
hasBeenSent	INTEGER
isValid	INTEGER
hopCount	INTEGER
signature	BLOB

Direct Outgoing Messages

Id	**INTEGER**
created_at	INTEGER
status	BLOB
recipientUser	TEXT
user	BLOB
isDisaster	INTEGER
hasBeenSent	INTEGER

Friends

Id	**INTEGER**
user	TEXT
modulus	TEXT
exponent	TEXT
isDisasterFriend	INTEGER
isFollower	INTEGER
isFollowed	INTEGER

Mentions

Id	**INTEGER**
created_at	INTEGER
status	TEXT
userCode	INTEGER
isDisaster	INTEGER

Direct Messages

Id	**INTEGER**
created_at	INTEGER
status	TEXT
userCode	INTEGER
isDisaster	INTEGER

Search Results

Id	**INTEGER**
created_at	INTEGER
status	TEXT
userCode	INTEGER
isDisaster	INTEGER

Figure 5.13.: Database tables

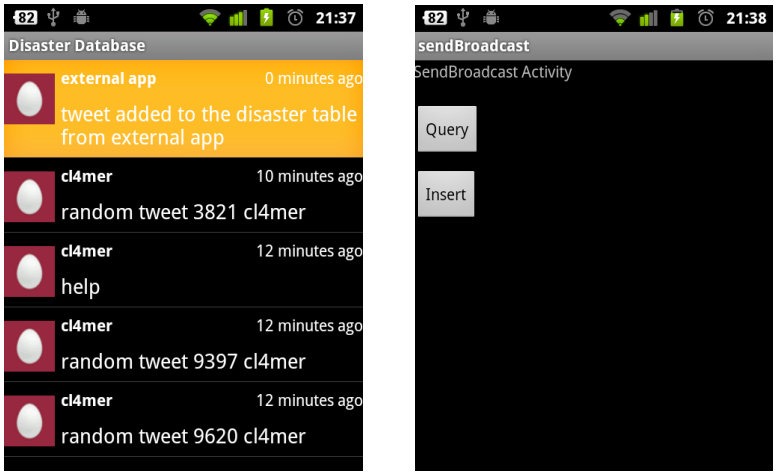
- Timeline: this table stores the tweets that are shown in the main user interface. Along with the author identification code there are two binary fields used to distinguish disaster and favorite tweets.
- Favorites: this table stores the favorite tweets of the authenticating user.
- Addresses: this table is used during disaster mode to store the list of peers the device has met. It is necessary to know the timestamp of the last meeting and the number of tweets at that time.
- Profile Pictures: here the users profile pictures are stored.
- Disaster: this table stores all the disaster tweets. Each one is identified by a local Id and a timestamp. It is necessary to store both creation and reception time since it will be necessary to sort them when delivering to other peers. In order to reduce the flooding in the network, the last hop is stored to avoid sending it back. Since retweets are spread in the ad hoc network as well, they must be identified with a binary field (isFromServer). Every tweet has a signature and a field that indicates if it has been published on the central servers.
- Direct Outgoing Messages: stores the encrypted outgoing messages. Moreover, also the messages a device is carrying for other users are stored.
- Friends: in this table all the authenticating user's followers and the users followed by him are stored. Also, the peers met opportunistically are stored here, along with the public keys

used for encryption and/or digital signature.

- Mentions: this table stores all the tweets in which the authenticating user has been mentioned.
- Direct Messages: this table stores the direct messages received. The field "isDisaster" is necessary to differentiate between normal and disaster messages that have been received opportunistically.
- Search Results: this table is used as a temporary buffer for the search results, especially in case of a disaster search.

5.5.2.2. Content Provider

Since Twimight is meant to be integrated with other external applications, a simple content provider has been implemented. It overrides some methods of the "ContentProvider" class, namely: query, insert, update and delete. The content provider allows to access the disaster table from external applications installed in the Android phone. It is the only way to share a database among applications. An external app needs to call a "ContentResolver" object which identifies the content provider that is the target of the request. Sharing tables of the database might be useful for instance for an application that needs to read all the tweets for text processing purposes or for the sensor application described in chapter 8. Figure 5.14 shows a simple external application that can insert tweets or query the disaster table.



(a) tweet added (b) External app prototype

Figure 5.14.: Tweet added by external app

5.6. Connection

This section explains in detail the reasons behind choosing Bluetooth as a communication technology as well as how the neighbor discovery works. Finally the communication protocol used for the data exchange is discussed.

5.6.1. Ad hoc Communication

In order to have opportunistic communications was important to choose carefully which underlying communication standard to use. Basically there were three possibilities: WiFi ad hoc, WiFi-Opp [36] and Bluetooth. WiFi ad-hoc would have been the best standard to be used if we just think about the protocol characteristics: no pairing, large bandwidth and broadcasting of datagrams. Unfortunately, it is not supported by stock Android phones and moreover it drains the battery too rapidly. Since the final purpose of this project was to publish the application on the Android market, the idea of rooting phones and installing custom network drivers has been discarded. Only a few experts would have been able to download and install the application. An alternative could have been the WiFi-Opp, which allows opportunistic communication leveraging the mobile access point feature (tethering), as well as static access points nearby. Unfortunately, a stable implementation was not ready when writing this thesis, but it might be integrated in the future as explained in the future work chapter. Hence the last option was Bluetooth. In spite of some limitations it has, like the cumbersome pairing, discoverability timeouts and limited ranges it was the technology used to implement Twimight. After all it offered the best trade off between battery lifetime (see section 6.2) and service provided in order to enable ad-hoc communications among Android phones.

5.6.2. Neighbor Discovery

A neighbor discovery mechanism has been implemented for Twimight in order to detect devices in communication range. For this purpose a background Thread periodically accesses the Bluetooth radio functionalities and activates the scanning operation. The results are broadcasted inside the Android system and received by a listener implemented in the main disaster Service. To avoid wasting of both energy and time, the devices are filtered and only the smartphones [2] are selected for a connection attempt. The scanning interval has been set to 120 ± 20 seconds. This represents a good trade off between energy consumption and delays. In fact, the higher the scanning interval the higher is the probability to miss some short encounters among mobile devices. The interval has been randomized in order to avoid scanning collisions and to decrease the probability to have several devices scanning at the same time in case of a densely populated network. In addition, if the device is still connecting to a peer when the discovery is launched, Twimight automatically delays the discovery for further 20 seconds.
A successful conection attempt may take up to 5 seconds [3] otherwise it drops automatically and starts connecting to another peer if any was detected. Sometimes, devices might attempt a connection simultaneously, so the attempt might take even 30 seconds. In order to avoid this, Twimight detects if the attempt takes more than 5 seconds and drops the connection anyway after 8 seconds. Since devices might stay in communication range for periods longer than the scanning interval,

[2] such as Android phones or Apple iPhones
[3] value obtained experimentally

they will connect to each other and exchange information only if they have not communicated in the last two minutes. This is a simple but effective way to additionally reduce connection collisions and battery draining, since it is unlikely to see users sending a tweet every minute.

5.6.2.1. Battery saving

In order to extend the battery lifetime, a simple mechanism has been designed (See section 6.2 for more information). Once the disaster mode has been activated, a broadcast listener is initialized. It periodically listens for the battery level and it takes the following actions:

- If the battery level is lower than 50% but higher than 30% it halves the discovery frequency.
- If the level is lower than 30% the discovery process is disabled and the device waits passively for incoming connections.

5.6.3. Data Exchange

Whenever two nodes are in communication range and connected, they exchange a couple of packets according to a simple communication protocol shown in figure 5.15. It allows them to exchange all the relevant information stored in the local databases. The first message sent is a "Hello" packet. Essentially, it is used to send the timestamp of the most recent tweet belonging to the connected peer in case it is the first meeting. That is because a node might already have some tweets belonging to the peer, even at the first encounter. Sending the timestamp can help to decrease the amount of tweets sent, hence reducing battery draining and connection time.

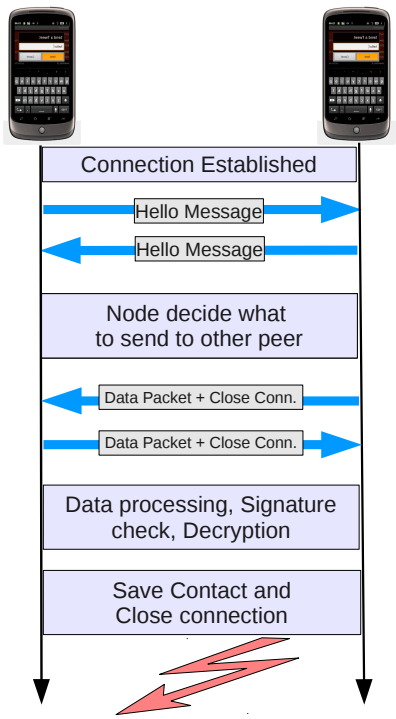


Figure 5.15.: Message Flow between two nodes

In case of successive encounters, the "Hello" message is used to send the number of tweets seen for the connected peer in the previous encounters. It allows to optimize further the epidemic spreading. After receiving the Hello packet the nodes start preparing the Data for the other device. Then, the packet is serialized by an Object Output Stream and sent over the radio channel using the Bluetooth socket. The received packet is processed by the devices in order to extract all the tweets and direct messages that it is carrying. Then, they are added to the local database. Last actions performed are saving the contact into the database, so that it can be used for further encounters and closing the connection. In case something unexpected happens, the connection is closed anyway by default after 8 seconds. The following subsections describe in more detail all the decisions and operations performed by Twimight during the data exchange phase.

5.6.3.1. Hello phase

Figure 5.16 shows how Twimight works once a connection has been established. Basically, if two nodes have not met before, the application scans the local database in order to find whether it has already some tweets belonging to the connected peer. In fact, they might have been delivered opportunistically from other peers in the ad hoc network. In case it finds some, the most recent creation timestamp will be sent, otherwise it will be set to zero. If two peers have already met in the past, this means that they might have exchanged some tweets. Therefore, each peer sends the number of tweets he has seen in the previous encounter.

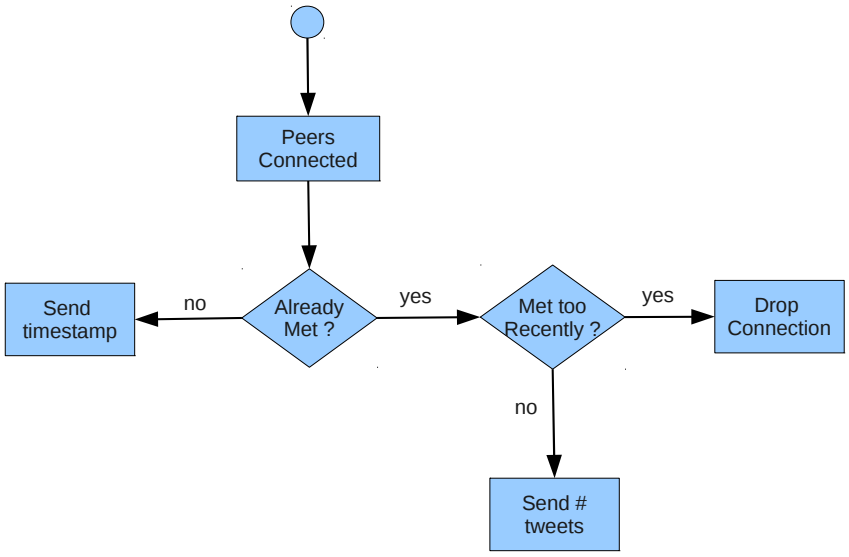


Figure 5.16.: Sending an Hello Packet

Once a peer receives an Hello packet, it is processed in order to decide which information has to be sent to the other peer. According to the information received, only the new one will be sent so that the overhead of the epidemic spreading is reduced. Upon receiving an Hello packet, Twimight checks if it is carrying the number of messages or a timestamp. If the packet represents a first encounter, it extracts the timestamp that is used to determine which are the new packet to be sent. On the contrary if the packet is a successive encounter one the number of tweets is extracted and then used to compute the list of new tweets to be sent to the connected peer. If there are no new tweets, a closing request is sent by the client. The process is shown in figure 5.17.

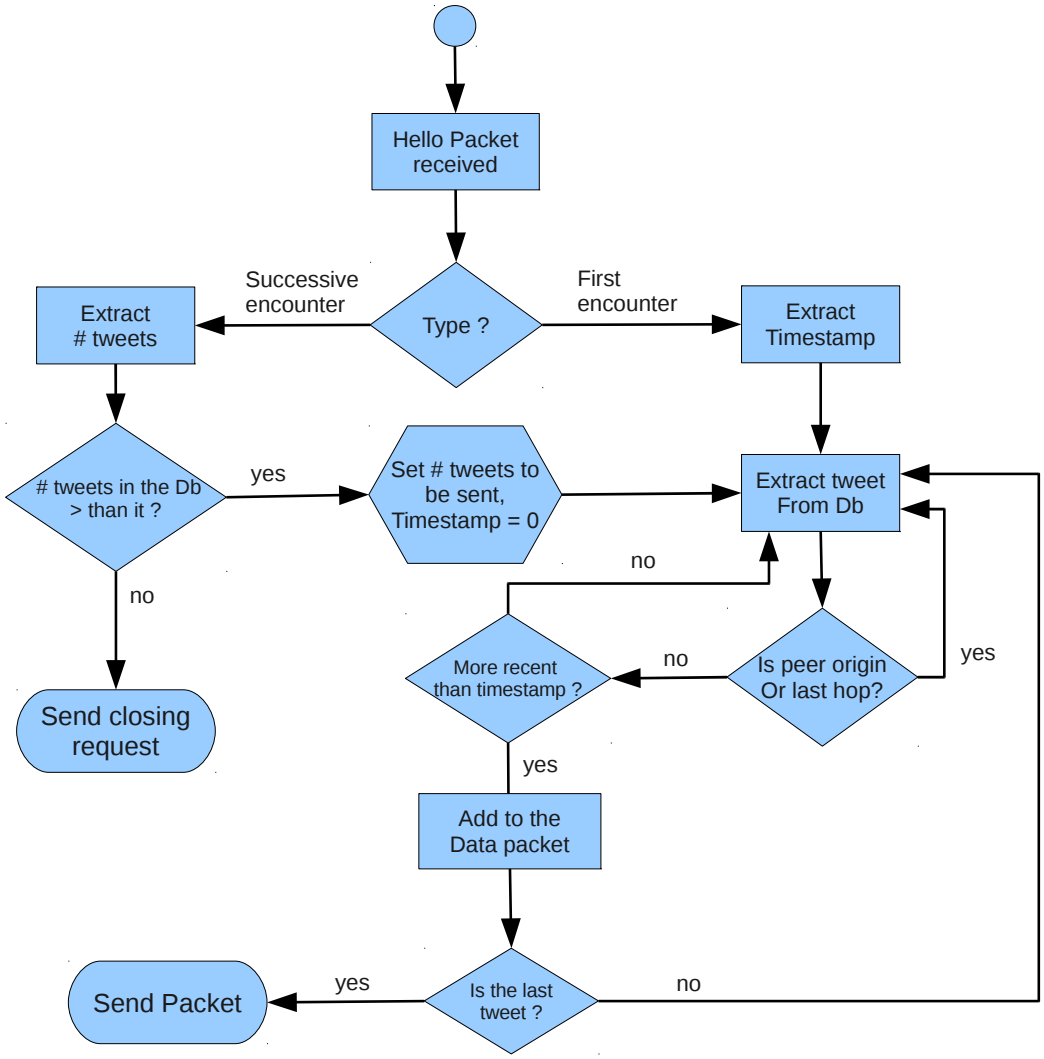


Figure 5.17.: Receiving Hello packet

5.6.3.2. Data phase

After the Hello phase, the data part starts as explained in section 5.6.3. Here, basically the received packet is processed in order to extract all the tweets and direct messages that is carrying. For the sake of simplicity, two different flowcharts are shown, even though the processing is performed in the same function. The processing operations for the tweets are explained first. Once Twimight receives a data packet, it extracts all the tweets one by one. For every tweet, a digital signature verification is performed (see section 7.3). If it succeeds the tweet is added to the disaster table. The sender information and the respective public key is stored in the Friends table. After it has been successfully added, an Intent is broadcasted into the system in order to refresh the main user interface. All the operations are shown in figure 5.18.

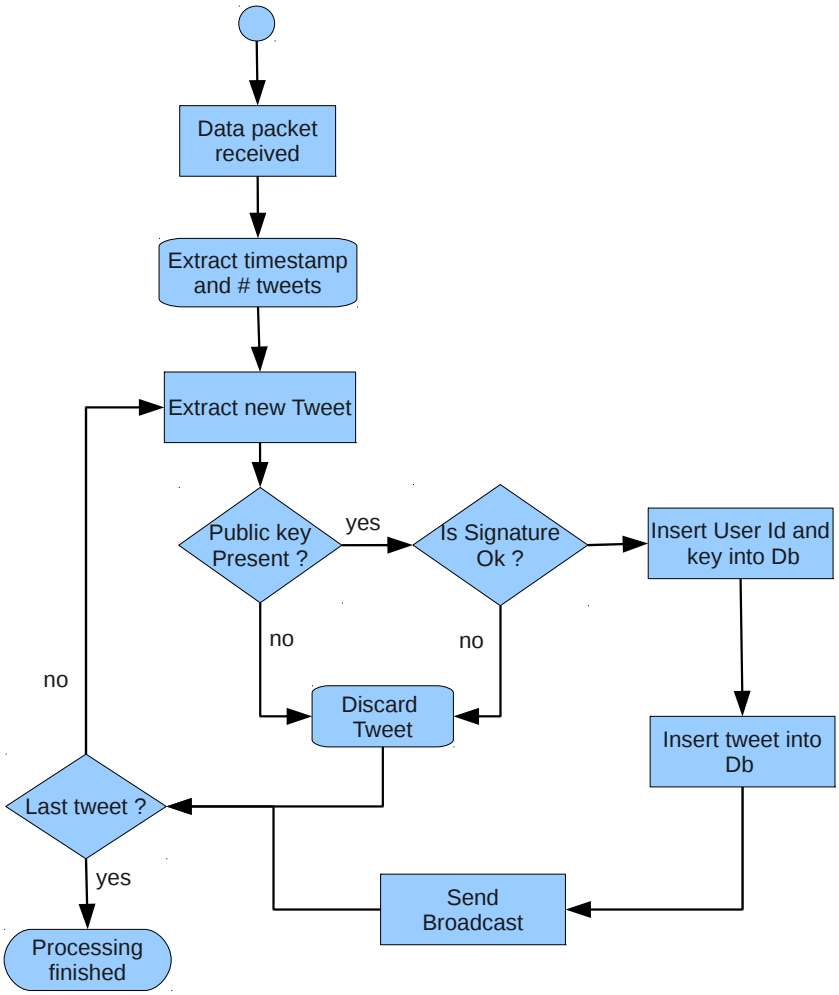


Figure 5.18.: Tweets received flowchart

A similar processing is carried out for the direct messages. Every message is extracted and for each one Twimight checks whether is the user is the recipient of the message or not. If it is not the recipient, it just inserts the encrypted message into the Outgoing table. On the contrary it decrypts the message with his private key and inserts into the normal direct messages table. The process is shown in figure 5.19

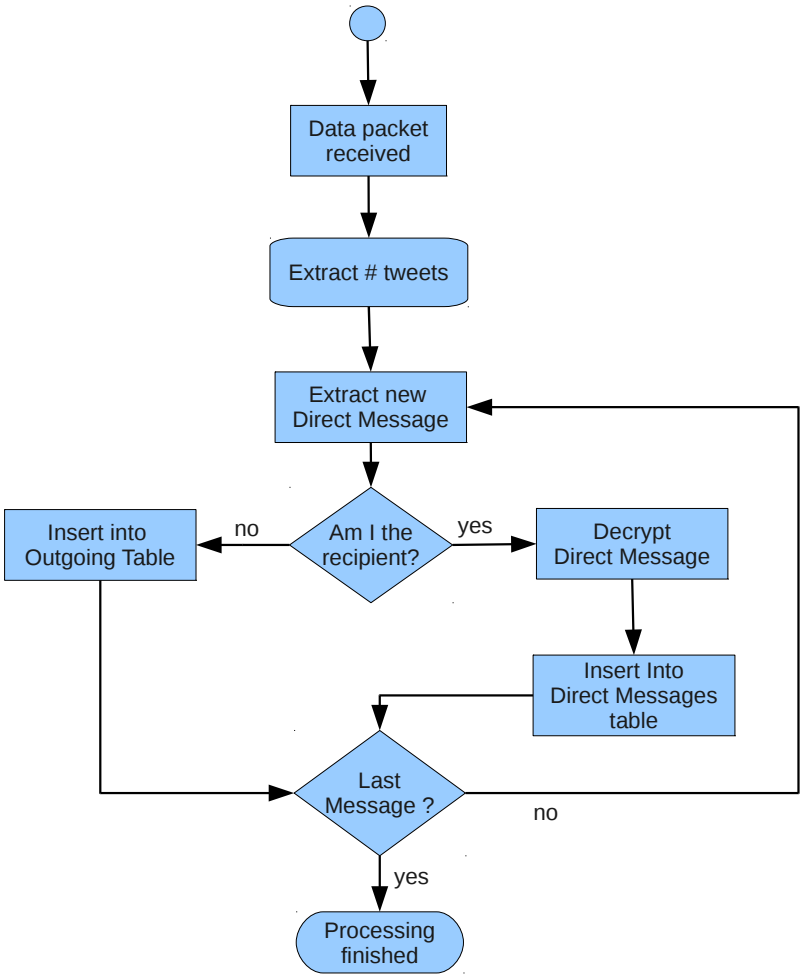


Figure 5.19.: Direct Messages received flowchart

6. Experiment and Results

In order to evaluate the performance of Twimight with realistic mobility patterns, multiple experiments have been carried out at the ETZ building of the ETH Zurich. The mobility patterns and the interactions reflect a typical office scenario. The experiments involved 5 people, four at the G floor and one at the F floor. Each one had to carry the device during the day of normal working. The most important experiment lasted 2 days and 915 tweets were generated. The majority of them were generated automatically by a background thread in order to require the least user interaction. It means also that some tweets were generated during the night. They have been taken into account separately, and almost all of them were recognized and filtered in the post processing phase. The devices used were Nexus One phones running Android 2.3.3. Figure 6.1 shows the phones used for the experiments.

Figure 6.1.: Nexus One phones for the experiment

6.1. Performance Evaluation

The following subsections will explain the results obtained, namely: delivery delay, delivery ratio and hop count.

6.1.1. Delivery Delay

The first result shown represents the delivery delay CDF for the messages. It is an important metric used to evaluate opportunistic network and routing protocols performance. It represents the difference between the time instant when a message is delivered and the creation time. In order to reduce the effect of clock drifts, the accuracy has been chosen to minutes. In this way some seconds of difference among the internal clocks can be tolerated. The results are shown in figure 6.2.

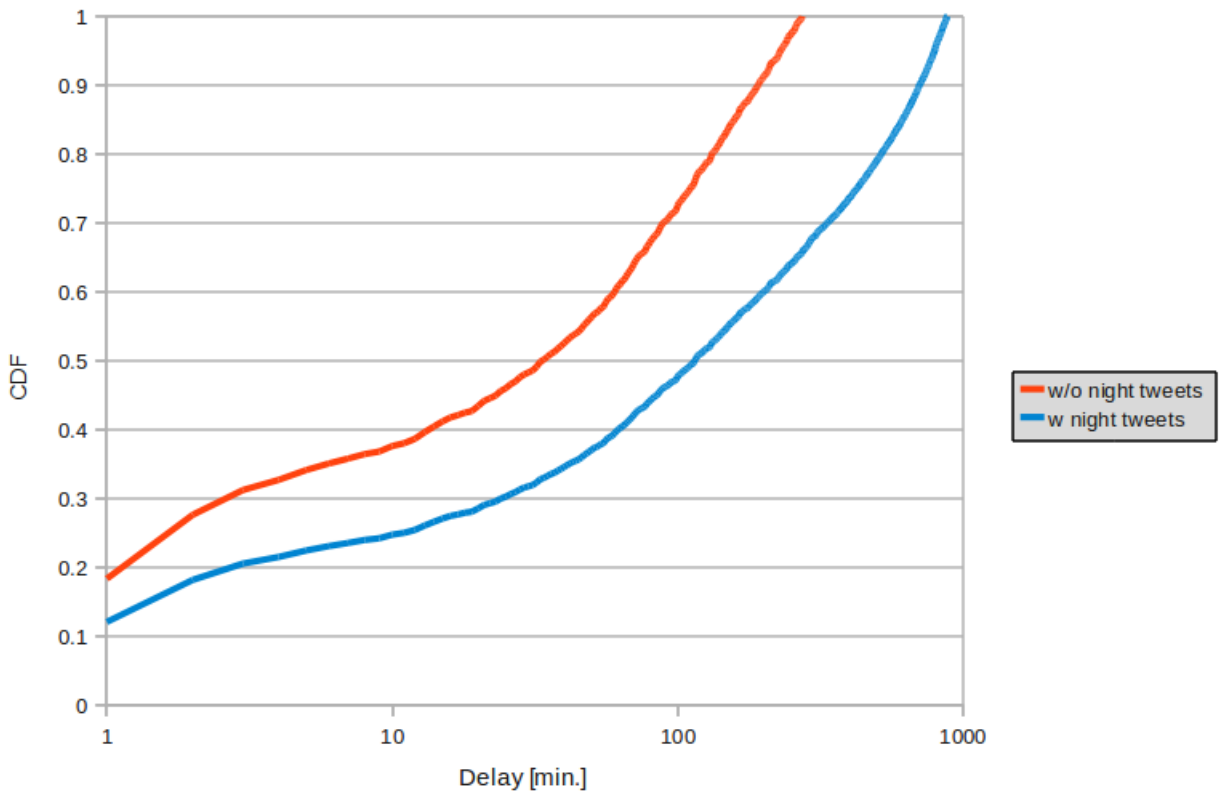


Figure 6.2.: Delivery Delay CDF

From the figure is possible to notice that if we discard the tweets generated during the night, around 30% of the messages have been delivered within 4 minutes. This is clearly a consequence of clustering since two pairs of phones were often in communication range. After 100 minutes 70% of the messages have been delivered. Evaluating the slope of the curve over time, is possible to deduce some peculiarity of the mobility model. Basically, the majority of the tweets are delivered either in the first 5-6 minutes or after 30 minutes and it could be used to deduce how often people met during the experiment.

6.1.2. Delivery Ratio

The delivery ratio R_j to a user j represents the fraction of the messages that were delivered to j over the total number of messages generated in the network during the experiment. It can be seen in the table 6.1 the delivery ratio for each device involved in the experiment. Almost all of them are above 80% except device 1. The reason of this can be explained since this device was located in another floor of the building, therefore had less opportunities to meet peers. The average delivery ratio was hence 79.7%.

	device 1	device 2	device 3	device 4	device 5
Delivery ratio	0.577	0.871	0.8066	0.9071	0.8251

Table 6.1.: Delivery Ratio

6.1.3. Hop Count

Another important parameter used to evaluate opportunistic networks is the hop count distribution. In the experiment performed 51% of the tweets were forwarded over multiple hops, showing that Twimight gives a good multihop dissemination service. Results can be seen in figure 6.3.

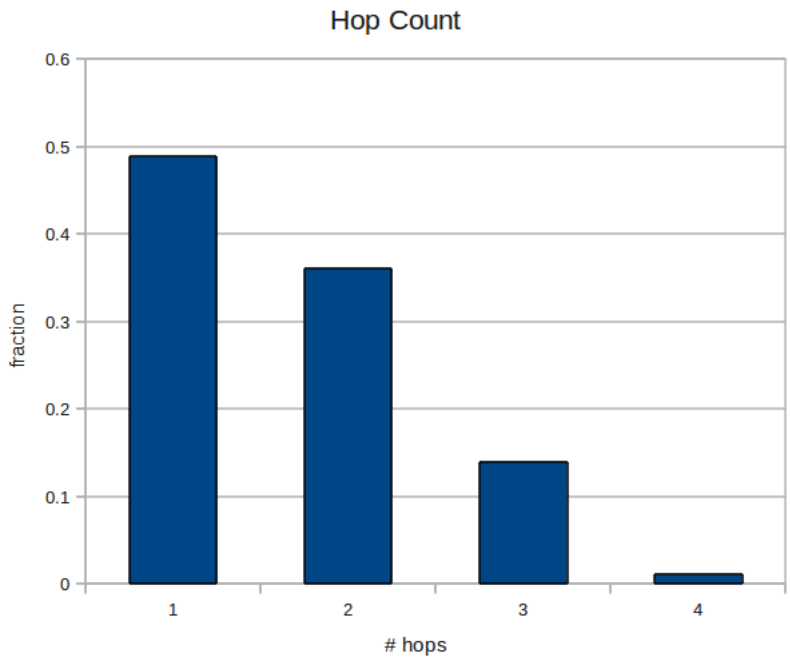


Figure 6.3.: Hop Count

6.2. Power Consumption

An important parameter for the experiment and in general for mobile devices, is the battery lifetime. For this reason, it was necessary to evaluate the impact of Twimight in disaster mode on the phone battery. Multiple tests have been carried out in order to evaluate the impact of different scanning frequencies and finally evaluate the impact that WiFi would have on the system. The basic test setup in order to have consistent results was the following:

- Sim card present and good GSM signal strength.
- Static Brightness for the display.
- No phone calls.
- No heavy applications running.
- Turn on the display only for a couple of minutes per day.

In order to evaluate qualitatively the difference between the phone in disaster mode and the normal battery lifetime it is possibile to see the plots provided by the Android system for a single isolated mobile phone with scanning interval of 1 minute (worst case). It is shown in the figure 6.4.

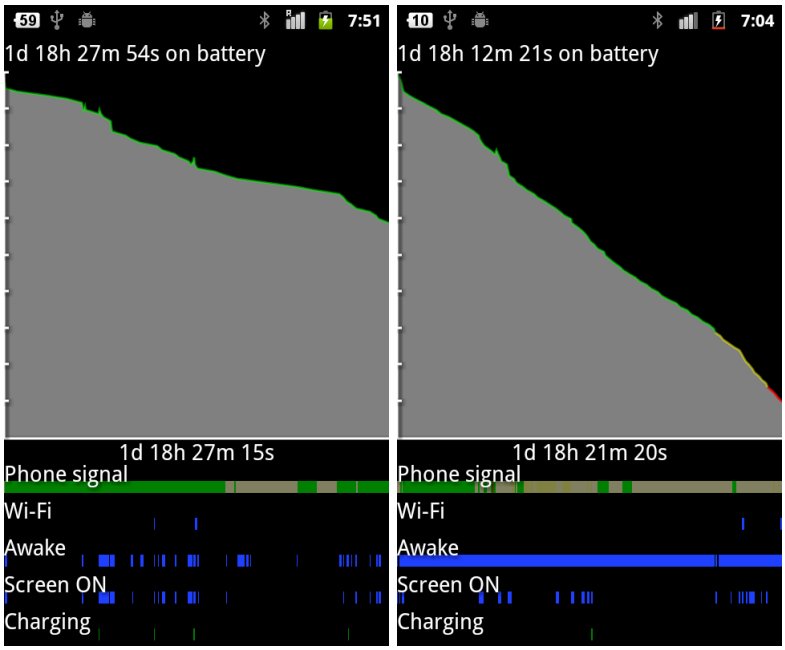


Figure 6.4.: Device screenshots

It is clear that the battery drains faster with the disaster mode active with the highest scanning frequency, however the lifetime (time to reach 10% charge) has been 1 day and 18 hours, that is pretty reasonable for a disaster scenario. The lifetime without the disaster mode active was 3 days and 10 hours almost the double of the case with disaster mode active.

For a more accurate quantitative evaluation, it was necessary to sample the battery level periodically implementing a broadcast receiver. Different scanning frequencies have been tested, namely 1,2 and 4 minutes. Moreover, also the impact that WiFi ad hoc would have on the systems has been evaluated experimentally.

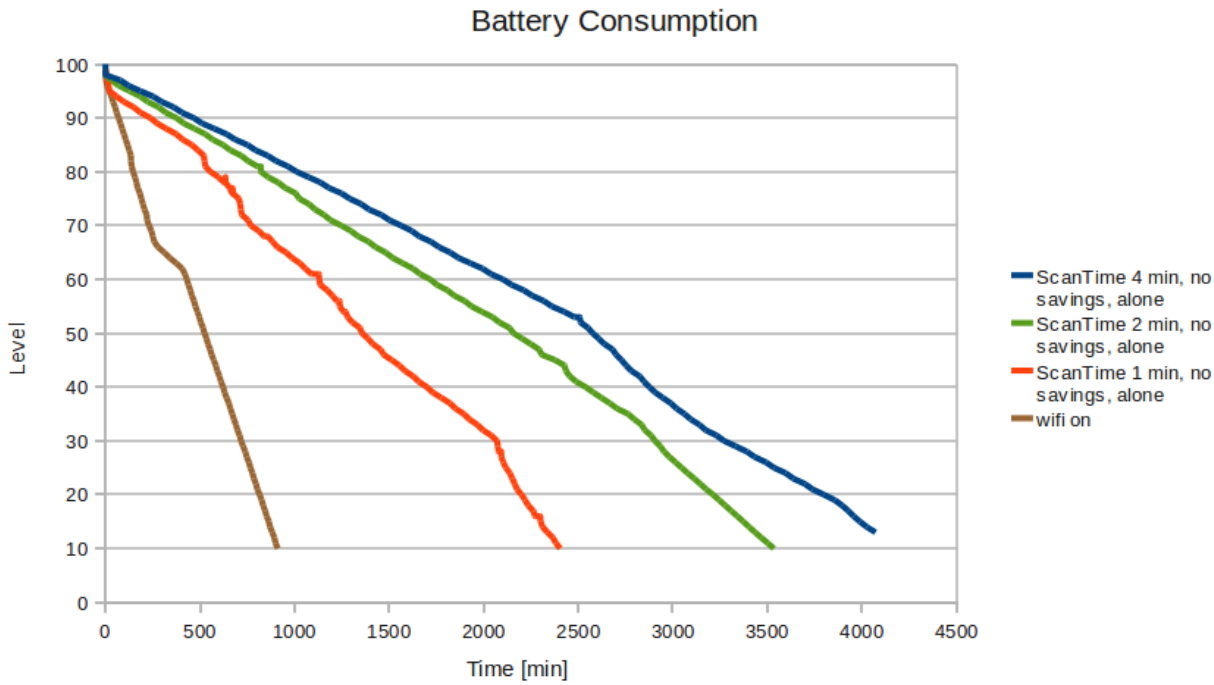


Figure 6.5.: Impact of different configurations

The worst performance as expected has been with WiFi enabled as shown in figure 6.5. The battery lifetime was limited to just 16 hours. Better results were achieved with the Bluetooth standard. The scanning frequency clearly had an impact on the battery duration, and the lowest registered was around 41 hours scanning every minute. Decreasing the scanning interval to 2 minutes, the lifetime increased significantly to 58 hours. Finally scanning every 4 minutes, the device can be carried around without charging up to 2 days and 20 hours. The scanning interval of 2 minutes provided the best trade-off in terms of battery lifetime and possibility to catch short encounters as well. For this reason we decided to use this value for the prototype implementation.

7. Security Architecture

Usually a natural disaster is not the enviroment where sophisticated attacks could take place. However, it was important to build the basic structure in order to support security features. Opportunistic communication security requirements are a bit different from the normal client-server architecture. Nodes should be able to verify the identity of the sender as well as reduce effect of spam on the network. Some of the security features for Twimight are meant to be implemented or completed in the future, but it has been a great achievment for this work to start implementing some of them. The main assumption for the architecture is that the security elements must be set up when the clients authenticate and therefore before loss of connectivity occurs. When connectivity is lost Twimight has to be able to work in a complete distributed way. The set up phase will be performed using a server. The drawback of this architecture is that a user cannot join the network during a disaster. This is a tolerable trade-off for a system which is much more simple and easy to use than a completely distributed one.

7.1. Requirements

The main security requirements for Twimight are the following:

- Guarantee authenticity of the disaster Tweets in order to prevent someone to fake his identity.
- Disaster Direct Messages need do be encrypted since they are spread epidemically and only the actual recipient has to be able to read them.
- Limit the impact of both malicious and naive spammers inside the network .

7.2. Public Key Infrastructure

In order to support security it is necessary to introduce a Twitter Disaster Server (TDS). It acts basically as a certificate authority for the public keys of the clients and in addition it will used to distribute them to the devices since the spreading in the opportunistic network might be too slow in some scenarios. The root certificate will be included in the next versions of Twimight. Figure 7.1 shows the prototype of the security architecture to be provided.

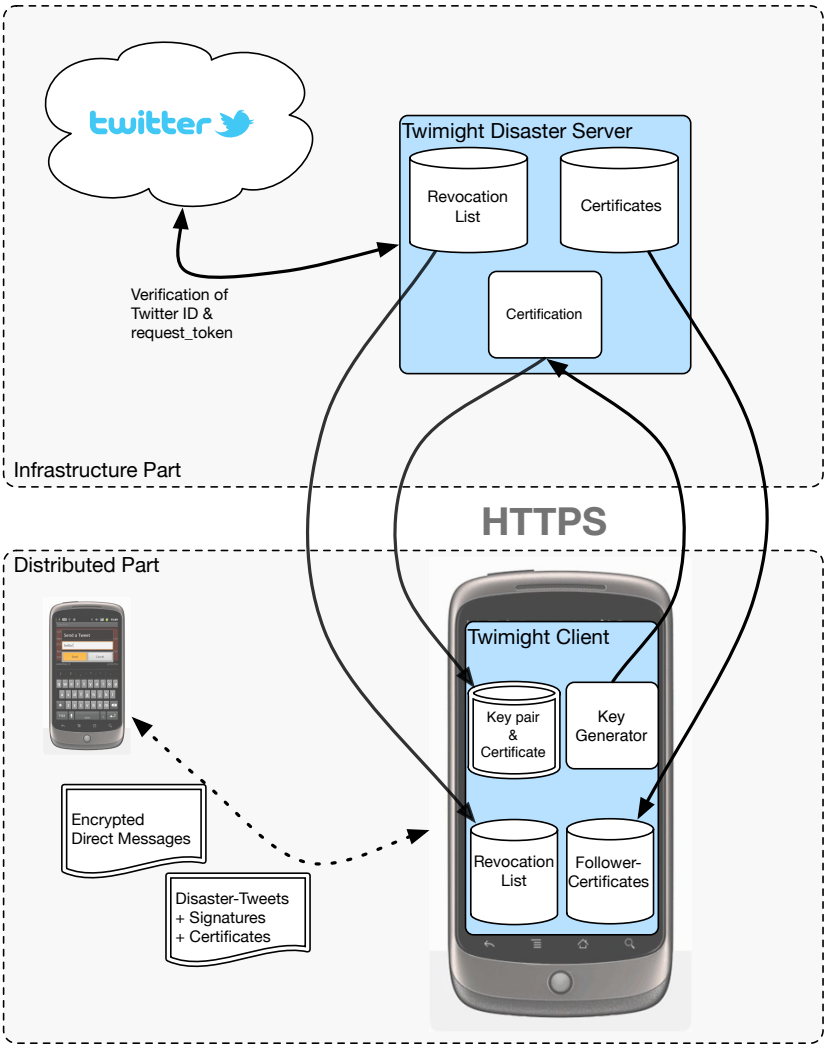


Figure 7.1.: Security Architecture

Communication with the TDS is encrypted using the HTTPS protocol. For key generation and encryption functionalities two algorithms can be used in general,namely: RSA [37] and Elliptic Curve Cryptography (ECC) [38]. The latter is generally more efficient than RSA [39] since it needs smaller keys, however neither Java nor Android supports this system. This is the main reason for using the RSA algorithm. Twimight generates public/private key pairs locally and stores them in a SharedPreference file. Then sends them with an HTTP POST method to the TDS along with the OAuth access token and secret. The key generation takes place at the first login into Twitter and also periodically every two days. The keys sent to the TDS server need to be authenticated therefore it sends back to the client the certificate for that public key. It will be

verified with the root certificate that will be installed in the next versions of Twimight. In order to obtain all the necessary public keys there are two ways: get them from the TDS and from the received messages in the opportunistic network.

7.3. Tweets Authenticity

Once in disaster mode Twimight need to verify the authenticity of the tweets received. Every client uses its own private key to digitally sign the tweets that are to be sent epidemically to the opportunistic network. The tweets are signed at creation time [1] and added to the disaster table in the local SQLite database. Upon reception, the signature of every tweet is checked as explained in the section Data Exchange 5.6.3.2. If the verification process fails, the tweet is simply discarded.

7.4. Direct Message Encryption

Since direct messages contain private information, they need to be protected somehow when they are spread in the opportunistic network. Basically, Twimight encrypt every direct message with the recipient's public key before the message is forwarded from device to device. Only the legitimate recipient will be able to decrypt the messages since it is the only one that has the corresponding private key. In order to use this scheme, it is necessary of course to have the all the public keys when the network breaks.

7.5. Spam Control

In order to prevent a spammer from attempting a Denial of Service attack by flooding the network with useless tweets, we implemented a simple spam control mechanism. Basically the number of tweets exchanged in one communication round is limited to 500. If a device receives a packet with more tweets it will automatically discarded. Out of the 500 tweets half of them are reserved for the authenticating user, leaving capacity for legitimate communication.

[1] Signature is computed over text, tweet Id, username, user Id and timestamp

8. Sensor Data over Twimight

Twimight has been extended in order to automatically spread sensor data when in disaster mode. Sensor readings in fact can be translated into tweets without any user interaction. They might be extremely useful to give more information about the user's environment as well as providing a sign of life. Up to date, the the sensors used are the ones available on the Android mobile phone, namely accelerometers, GPS sensor, light sensor. The readings, are aggregated and formatted as tweets. The sensing part has been implemented in a separate application, developed at the Communication Research Group at the Uppsala University. The communication between the background application and Twimight is done broadcasting intents into the system. Twimight therefore has a listener to catch them and extract the information they are carrying. The sensing architecture can be seen in figure 9.1.

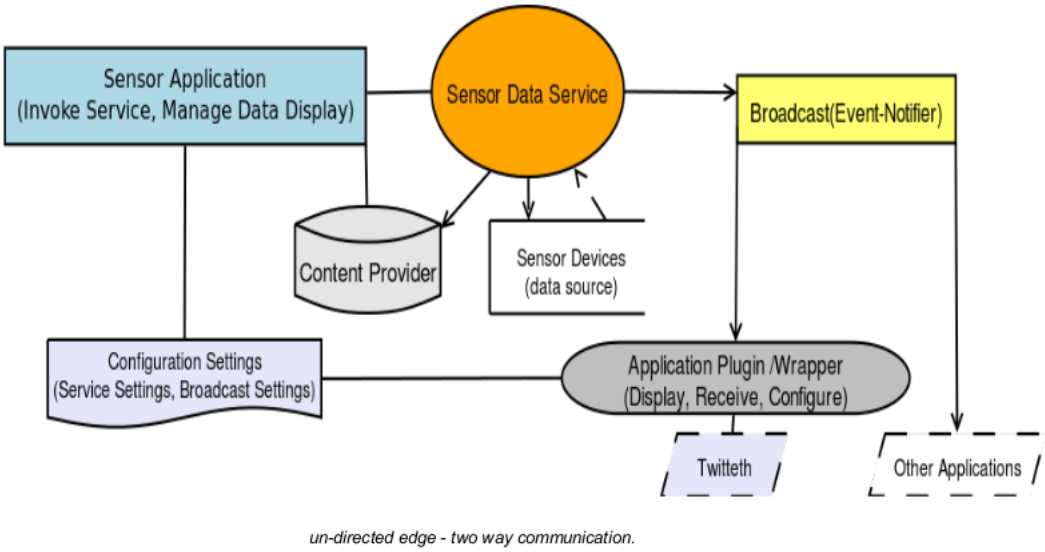


Figure 8.1.: Communication between sensors and Twimight

In order to enforce security some custom permissions have been defined. Basically they are used for two reasons:

- Allow only the background sensing application to broadcast data into the system.
- Authorize Twimight to receive them.

It represents a good security feature since it is really unlikely to have malicious applications on the same phone that know which kind of permissions are being used.

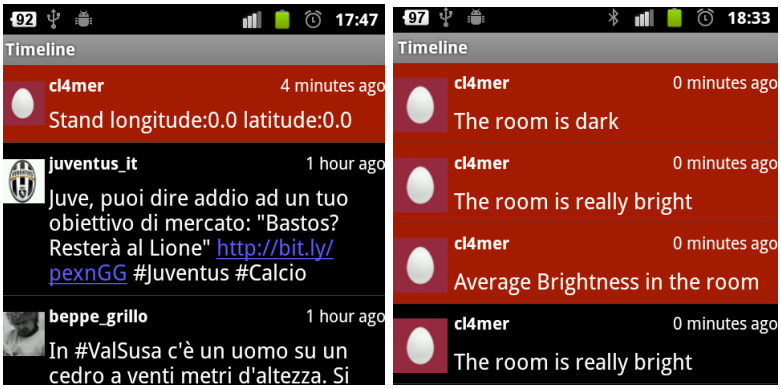


Figure 8.2.: Sensor readings examples

9. Conclusions and Future Work

In this master thesis a disaster ready Twitter client for Android phones has been implemented (Twimight). The main purpose of this application is extend the functionalities of Twitter clients available in almost all new smartphones. Online social networks and especially Twitter have been used largely in case of disastrous events and political uprisings, however the infrastructure might break down or can be intentionally blocked, making them almost useless. In fact, they were not designed to support those kind of events, therefore a new way to allow communication in such situations has to be found. The one we proposed in this thesis has been to exploit the short radio technologies embedded in actual mobile phones in order to communicate without the need of instrastructure in a opportunistic fashion. It could help a lot both in case of natural disasters and in order to circumvent censorship of dictatorial governments. Upon enabling the disaster mode the smartphones will start working in a hybrid mode, in which peer to peer wireless communications take place as well as attempting to access the infrastructure to publish the gathered information. Up to date the ad hoc wireless technology used is Bluetooth. In spite of the cumbersome pairing, it allowed good battery lifetime and it did not need rooted phones. It is really important to gather a consistent user base, since only a few expert users root their own devices.

A disaster scenario is not an environment prone to sophisticated attacks, however a political uprising might be. In order to cope with that, some basic security features have been designed and implemented. Some of them have to be completed and improved as explained in the future work section. An interesting part of this project has been the integration of Twimight with a Sensor application developed at the University of Uppsala. Sensor readings in fact could help a lot in order to provide additional information about a user's environment as well as providing a sign of life. Additional different applications might need to communicate with Twimight. This is feasible since a simple interface has been implemented.

For the Future Work, we propose the following:

- Simulate and analyze different dissemination strategies in order to evaluate the overhead of flooding.
- Limit the spreading to a certain geographic area or to some interest groups (via #hashtags).
- Extend and test Twimight with different radio communication standards (WiFi ad hoc and WiFi Opp).
- Improve the security architecture implemented in this thesis.
- Integrate Twimight with a CNA based routing algorithm for DTNs.
- Allow the Twimight Disaster Server to publish information on behalf of users.

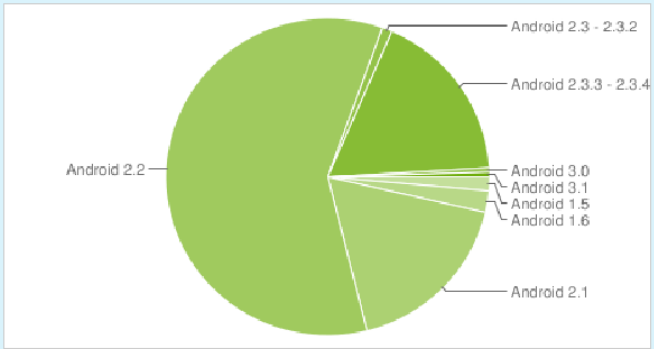
A. User How-to

This appendix explains the basic requirements to run the application as well as some hints in order to install it both from the Android market or the SDK.

A.1. Requirements

The application has been developed for Android phones. To use it effectively the basic requirements for the smartphone are:

- Android phones running at least version 2.1(Eclair) in case of secure RFCOMM Bluetooth socket.
- Android phones running at least version 2.3.3(Gingerbread) in case of insecure RFCOMM Bluetooth socket (No Pairing).
- Support Bluetooth
- 3G or WiFi during normal mode
- Support RSA Encryption
- At least 1 MB free space on the internal memory



Platform	API Level	Distribution
Android 1.5	3	1.4%
Android 1.6	4	2.2%
Android 2.1	7	17.5%
Android 2.2	8	59.4%
Android 2.3 - Android 2.3.2	9	1%
Android 2.3.3 - Android 2.3.4	10	17.6%
Android 3.0	11	0.4%
Android 3.1	12	0.5%

Data collected during a 14-day period ending on July 5, 2011

Figure A.1.: Android usage statistics [9]

Note that the first requirements do not represent a big problem since 95% of the phones are actually running a version equal or higher to 2.1. See Figure A.1.

A.2. Installation

There are two ways to install Android applications: from the apk file or directly from the market.

A.2.1. Installing Applications With Android SDK

This method is used for installing an app from an apk file and allows to avoid to use the Android Market. It is used mostly by developers.

- Install the Android SDK and the USB drivers
- Modify the Android settings in order to allow to install third party applications. Under "Settings," select "Application Settings" and then enable "Unknown Sources."
- Open a terminal window and type: adb install app_path/app_name.apk
- You are done. The app is installed

A.2.2. Installing from the Market

The market is the equivalent to the Apple's app store. It is the easiest way to install the application for normal users. Basically it is necessary just open the Market application from the menu[1],search the desired application and install it clicking the button on the right hand side of the screen.

[1] It is necessary to have internet connectivity

B. Developer How-to

This appendix shows the basic requirement to start developing Android applications and some tips in order to setup the system. The Twimight online repository is presented as well as the basic commands in order to download the source code. Finally some important background for developing on Android is given.

B.1. Requirements

The basic requirements in order to implement and test the application were:

- Eclipse IDE[1]
- Java SDK
- Android SDK [2]
- ADT plugin for Eclipse [3]
- OAuth Libraries
- Twitter APIs [4]
- At least two Android phones

The Java and Android SDK (Standard Development Kit) as well as the ADT (Android Development Tools) plugin were necessary to build the main structure of the application including the User Interface. The IDE used was Eclipse. It is a famous multi language development environment that can be easily extended with different plugins. The one needed for Android development is the ADT that is designed to give a rich and powerful environment to build such applications. In order to be able to login into Twitter the application needed to used the OAuth Signpost Libraries that provided the basic functionalities to implement the OAuth flow. After successful login the application communicates with the Twitter central servers using the Open Source JTwitter APIs.

B.2. Tips

Anyone who is interested to compile or run Twimight should observe the following:

- I strongly recommend using the Eclipse IDE (Integrated Development Environment) as Java Editor. You can download the last version from here[5].

[1] http://www.eclipse.org
[2] http://developer.android.com/sdk/installing.html
[3] http://developer.android.com/sdk/eclipse-adt.html
[4] http://www.winterwell.com/software/jtwitter/javadoc/index.html?winterwell/jtwitter/Twitter.html
[5] http://www.eclipse.org/downloads/

- Install the Android SDK and the ADT plugin [6] which are the standard environment to develop Android applications.
- Download the essential components for the Android SDK as well as setup the AVD Manager from the graphical UI of the SDK itself.
- If developing on Linux, install the USB cable drivers.
- Follow the Hello World tutorial [7].

B.3. Online Repository

Since the app is meant to be released as an Open Source software the code is available under GNU GPL v3 license on the Google Code online repository at the address: *http://code.google.com/p/twimight/*.

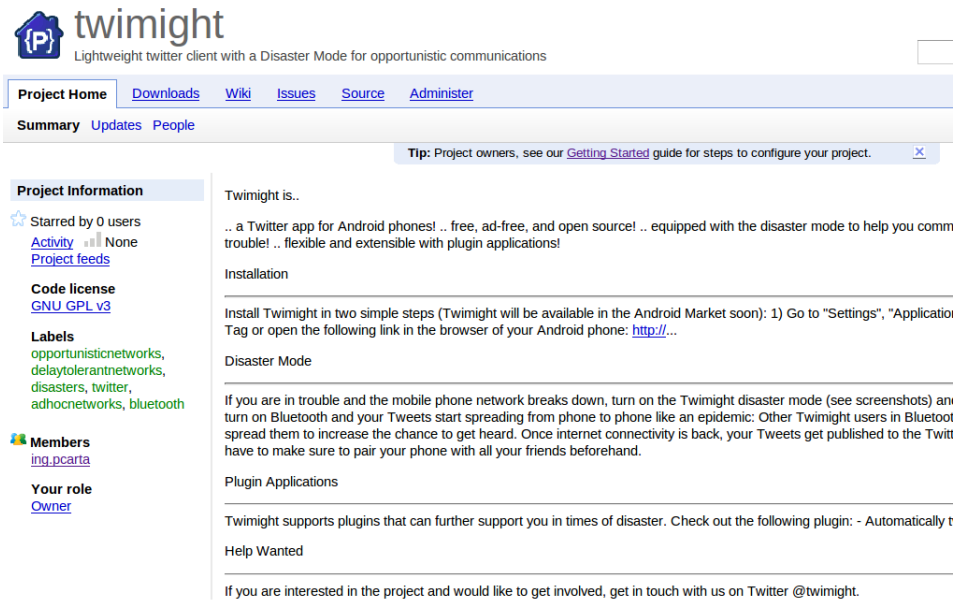


Figure B.1.: Online Repository

Whoever wants to download the code has to possibilities:

- Command-line access:
  # Non-members may check out a read-only working copy anonymously over HTTP.
  svn checkout http://twimight.googlecode.com/svn/trunk/ twimight-read-only
- GUI and IDE access
  This project's Subversion repository may be accessed using many different client programs and plug-ins [8].

[6] http://developer.android.com/sdk/installing.html
[7] http://developer.android.com/resources/tutorials/hello-world.html
[8] http://subversion.apache.org/packages.html

B.4. Fundamentals

Android applications are written in Java. The Android SDK is used compile the code into an Android package, which is a file with the .apk extension. Basically, it is the file that Android-powered devices use to install the application.

B.4.1. Components

- Activities: An activity represents a single user interface. Although the activities work together to form a complete application, each one is independent of the others. As such, a different application can start any one of these activities. An activity is implemented as a subclass of Activity .
- Services: A service is a component that runs in the background to execute some heavy or long operations. It does not provide any user interface. The most common examples are a background music player or simply a service that access the network. Another component, such as an activity, can start the service and let it run or bind to it in order to interact with it. A service is implemented as a subclass of Service
- Content Providers: A content provider allows to share application data. For instance it is possible to share a SQLite database, or any other persistent storage location. Through the content provider, other applications can query or even modify the data (if the content provider allows it).
- Broadcast receivers: A broadcast receiver is a component that listens for intents broadcasted in the system. A broadcast receiver is implemented as a subclass of BroadcastReceiver and each broadcast is delivered as an Intent object.

B.4.2. Intents

Activities, services, and broadcast receiver are activated by an Intent. It allows to bind individual components to each other at runtime. An intent is created with an Intent object, and it can be either explicit or implicit, respectively. For activities and services, an intent defines the action to perform. For example, an intent might convey a request to open another activity in the application. In some cases, you can start an activity to receive a result, in which case, the activity also returns the result in an Intent.

B.4.3. Activity Lifecycle

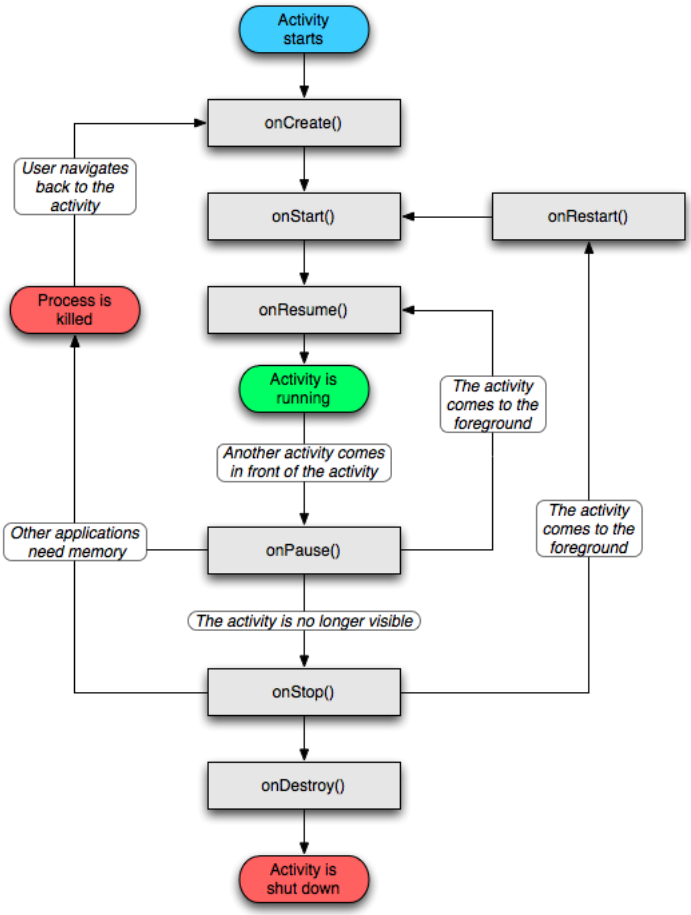


Figure B.2.: Activity Lifecycle source: http://developer.android.com/guide/topics/fundamentals/activities.html

B.4.4. The manifest

Every application has a manifest file. It stores the essential information about the application. Every component must be declared there along with all the permissions required to run. Moreover it stores the following information:

- The name of the Java package that uniquely identifies the application.
- The processes that will host application components.
- Declares the minimum level of the Android API that the application requires.
- It lists the libraries that the application uses.

C. Code

C.1. Android Manifest

This appendix shows the Twimight manifest :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:versionCode="1"
    android:versionName="1.0"
    package="ch.ethz.twimight">

    <application
        android:icon="@drawable/twitter_icon"
        android:label="@string/app_name" android:debuggable="true">

        <activity
            android:name=".MyTwitter"
            android:label="@string/app_name" android:launchMode="singleTask">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name="Prefs" android:configChanges="orientation|keyboardHidden"
                ></activity>

             <activity android:name=".OAUTH"
            android:label="@string/oauth_name" android:noHistory="true">
                    <intent-filter>
                <action android:name="android.intent.action.VIEW" />
                <category android:name="android.intent.category.DEFAULT" />
                <category android:name="android.intent.category.BROWSABLE" />
                <data android:scheme="my-app" android:host="twitt" />
            </intent-filter>
            </activity>

        <activity android:name=".Timeline" android:label="@string/timeline_label"
        android:noHistory="false"
          android:configChanges="orientation|keyboardHidden" android:launchMode="
              singleTask">
        <meta-data android:name="android.app.default_searchable"
                    android:value=".Search" />
        </activity>

        <activity android:name="Following" android:label="@string/friends_label"
            android:noHistory="false" >
```

```
42              <meta-data android:name="android.app.default_searchable"
43                     android:value=".Search" />
44          </activity>
45
46          <activity android:name="Followers"        android:noHistory="false" >
47               <meta-data android:name="android.app.default_searchable"
48                     android:value=".Search" />
49          </activity>
50
51          <activity android:name=".showDisasterDb" android:launchMode="singleTask">
52          </activity>
53
54          <activity android:name=".Mentions">
55               <meta-data android:name="android.app.default_searchable"
56                     android:value=".Search" />
57          </activity>
58
59          <service android:enabled="true"  android:name=".UpdaterService"  />
60          <service android:enabled="true"  android:name=".DevicesDiscovery" />
61          <service android:enabled="true" android:name=".RandomTweetGenerator"  />
62          <service android:enabled="true" android:name="ch.ethz.twimight.
                DisasterOperations" />
63
64          <activity android:name=".Search" android:launchMode="standard"
                android:configChanges="orientation|keyboardHidden">
65              <intent-filter>
66                   <action android:name="android.intent.action.SEARCH" />
67              </intent-filter>
68              <meta-data android:name="android.app.searchable"
69                     android:resource="@xml/searchable"/>
70          </activity>
71
72          <receiver android:name="ch.ethz.twimight.BootBroadReceiver"
                android:permission="android.permission.RECEIVE_BOOT_COMPLETED">
73              <intent-filter>
74                  <action android:name="android.intent.action.BOOT_COMPLETED" />
75                  <category android:name="android.intent.category.HOME" />
76              </intent-filter>
77      </receiver>
78          <activity android:name="ch.ethz.twimight.UserInfo" android:label="UserInfo"
79              android:configChanges="orientation|keyboardHidden"></activity>
80
81          <activity android:name="ch.ethz.twimight.DirectMessages" android:label="
                DirectMessages"
82              android:configChanges="orientation|keyboardHidden">
83          </activity>
84
85          <activity android:name="ch.ethz.twimight.SendDirectMessage" android:label="
                Send␣message"
86              android:configChanges="orientation|keyboardHidden" >
87          </activity>
88
89          <provider android:name="ch.ethz.twimight.MySuggestionProvider"
90                 android:authorities="ch.ethz.twimight" />
91
```

```
92          <provider android:name="ch.ethz.twimight.DbContentProvider"
93            android:authorities="ch.ethz.twimight.DbContentProvider"/>
94      </application>
95
96      <uses-sdk android:minSdkVersion="7" android:targetSdkVersion="10"/>
97
98        <permission android:description="@string/permDescription"
99              android:label="Receiving Permission"
100             android:name="com.permission.SENSOR"
101             android:protectionLevel="dangerous" />
102     <uses-permission android:name="android.permission.INTERNET"></uses-permission>
103     <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"></uses-
            permission>
104     <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
105         <uses-permission android:name="android.permission.BLUETOOTH" />
106         <uses-permission android:name="android.permission.VIBRATE" />
107      <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
108     <uses-permission android:name="android.permission.WAKE_LOCK"></uses-permission>
109     <uses-permission android:name="android.permission.READ_PHONE_STATE" />
110     <uses-permission android:name="android.permission.READ_LOGS" />
111         <uses-permission android:name="android.permission.RESTART_PACKAGES"></uses-
                permission>
112     <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED">/
            uses-permission>
113         <uses-permission android:name="com.permission.SENSOR"></uses-permission>
114
115     <uses-feature android:name="android.hardware.bluetooth"
116                   android:required="false" />
117     <uses-feature android:name="android.hardware.touchscreen" />
118 </manifest>
```

C.2. Disaster Mode main Classes and Methods

This appendix shows the most important methods and inner classes related to the disaster mode.

DisasterOperations Class

- **private void closeService()**
  It is called when the disaster mode is disabled. It stops the bluetooth sockets as well as unregister the broadcast receivers and release the wake lock used to maintain the CPU running during sleeping mode

- **class ShutDownDelayed implements Runnable{}**
  It is used in order to close the bluetooth sockets and respective threads with a bit of delay

- **private final BroadcastReceiver mReceiver{}**
  This broadcast receiver receives the results of the bluetooth scanning process and the discovery finished signal. For every device discovere it is added to the list in case it is a suitable smartphone.

- **Class CheckState extends AsyncTask<Void, Void, Void> { }**
  This class it us used for timeouts during the connection to another peer. If something goes wrong and no response is received the connection is dropped after the number of seconds specified by the constructor's parameter
  constructor parameter: int delay

- **Class ConnectionAttemptTimeout implements Runnable {}**
  This class it is used for detecting connection collisions (i.e. two devices attempt a connection simultaneously). In case a collision is detected the connection attempt is dropped.

- **private final Handler mHandler {}**
  The handler is used to communicate with the Bluetooth Threads. It can receive different kind of messages, namely: MESSAGE_READ, MESSAGE_DEVICE_NAME, MESSAGE_TOAST. The first kind of message contains data received from the bluetooth socket. The second is received when a new connection is active. The third kind of message is received when a connection attempt fails

- **private void sendTimestampOfLastTweet()**
  This function is called from the handler in order to send the timestamp of the most recent tweet contained in the database. If no tweets from the connected peer are available the timestamp is set to zero.

- **private void sendNumberOfTweets(String address, boolean timeout)**
  This function is called from the handler in order to send the number of tweets seen in the last encounter from the connected peer.
  parameter: String address is the MAC address of the connected peer.

- **private void extractDirectMessages(DirectMessage directMessage)**
  This method is used to extract a direct message. It verifies whether the user is the recipient or not. If the result is positive the message is decrypted and added to the direct messages

table, otherwise it will be added to the direct outgoint one.
parameter: DirectMessage directMessage, it represents a direct message data structure.

- **private void extractFields(SignedTweet readMessage)**
  This method is used to extract a Tweet. it verifies the signature and if it is correct the tweet is saved into the disaster table as well as copied into the timeline in order to be visualized in the main GUI.
  parameter: *SignedTweet readMessage, it represents a tweet data structure.*

- **private void processPacket(AbstractPacket ap)**
  This methos processes a received packet. Accordin to the type it takes different actions. If it is an Hello Packet it will extract the useful information and use it to determine which tweets have to be sent. In case of a data packet it will loop through the list of tweets/direct messages and pass them to the appropriate functions.
  parameter: *AbstractPacket ap, it represents the main abstract data structure from which Hello and Data packets can be derived*

- **private boolean didTheyMeetRecently(Cursor cursorPeers)**
  This method checks whether the connected peer has been met too recently.
  parameter: *Cursor cursorPeers, this is the cursor that contains the information about the connected peer.*

- **Class ConnectToDevices implements Runnable{}**
  This class is used in order to attempt a connection to devices in communication range. First of all it removes devices that have been met too recently from the list of neighbors, then if there is still connection it will attempt to connect to a peer and it will start a connection timeout.

- **Class SendClosingRequest Implements Runnable{}**
  This class is used to send a closing request(i.e. a packet with the closing connection flag set) in case there are no new data to be sent.

- **private Void SendRelevantTweets(Long Number, Long Time)**
  This method is called from the packet processing one. Basically it will scan all the disaster database in order to decide which tweets have to be sent according to the received parameters. First of all the tweets taken from the disaster database are ordered according to the time instant they were added to the local database. Then only the most recent are added to the outgoint packet and sent along with the direct messages.
  parameter*:* long number, it represents the number of tweets the connected peers has seen the last time from the client.
  parameter: long time, it represents the timestamp of the most recent tweet the connected peers has from the user.

- **private ArrayList<DirectMessage> getDirectMessages()**
  This method is used to get a list of direct messages from the local database.

- **private final BroadcastReceiver mBatInfoReceiver{}**
  This is the broadcast receiver used for the battery saving technique. If the level is too low savings techniques are applied.

Timeline Class

- **private void findMentionDisasterTweets()**
  This method scans the disaster database in order to find disaster tweets in which the authenticating user has been mentioned.

- **private void startServices()**
  This function is called once the disaster mode has been activated in order to start the background services, namely the devices discovery and the disaster operations.

- **private void publishDisasterTweets(boolean show)**
  This method is called periodically in order to attempt to publish the disaster tweets on the central servers. It will publish user generated tweets as well as disaster retweets.
  parameter: boolean show, it controls whether the user is notified with a toast message or not about the publishing operation.

- **class SendTask extends AsyncTask<Void, Void, Boolean> {}**
  This Async Task in case of disaster mode it will save the user generated tweet into the disaster table after digitally signing the tweet. In case there is no connectivity the tweet is copied in the timeline table as well in order to be visualized in the main GUI.

- **private final BroadcastReceiver airplaneModeReceiver {}**
  This broadcast receiver is activated once the disaster mode is enabled. In case the airplane mode is activated it will pause the disaster radio communications and the devices discovery in order to save energy. They will be restarted once the mode is disabled.

- **private final BroadcastReceiver externalAppReceiver {}**
  This broadcast receiver is activated when the disaster mode is enabled. It will listen for intents broadcasted by external applications like the sensing application developed at the University of Uppsala.

C.3. Example PHP script

Here a simple PHP script for the Twimight Disaster Server is shown. It allows to store all the public keys received from the clients.

```
<?php
$modulus = $_POST['modulus'];
$exponent = $_POST['exponent'];
$username = $_POST['username'];
$id = $_POST['id'];

$link = mysql_connect('localhost','root','') or die("Connection attempt failed");
mysql_select_db('twittethDb',$link);

//check whether id is empty
if (!empty($id) ) {

    //perform the query
    $query = "SELECT * FROM publicKeys WHERE id ='" . $id."'" ;
    $result = mysql_query($query,$link) or die("Error during the query" .
        mysql_error());

    //if modulus or exponent is not defined it is just  a query
    if( empty($modulus) || empty($exponent) )  {
            //check if it is already there
           if ( mysql_num_rows($result) == 1) {
               $row = mysql_fetch_assoc($result) ;
               print(json_encode($row));
           }
    }
    // if modulus or exponent are defined It is a call to insert them into the
        database
    else
        {

        // if there are rows it's an update of the key
        if (mysql_num_rows($result) > 0 ) {
            $query = "UPDATE publicKey SET modulus = '".$modulus."', exponent = '".
                $exponent."' WHERE id = '".$id."' ";
            $result = mysql_query($query,$link) or die("Error during the query" .
                mysql_error() );
        }
        //otherwise it's the first time I insert it
        else {
    $query = "INSERT INTO publicKeys(id,modulus,exponent,username) VALUES('".$id."
        ',".$modulus.",".$exponent.",'".$username."')";
            $result = mysql_query($query,$link) or die("Error during the query" .
                mysql_error());
        }
    }
}
    else
        die("Error");
mysql_close($link);
?>
```

D. Time Schedule

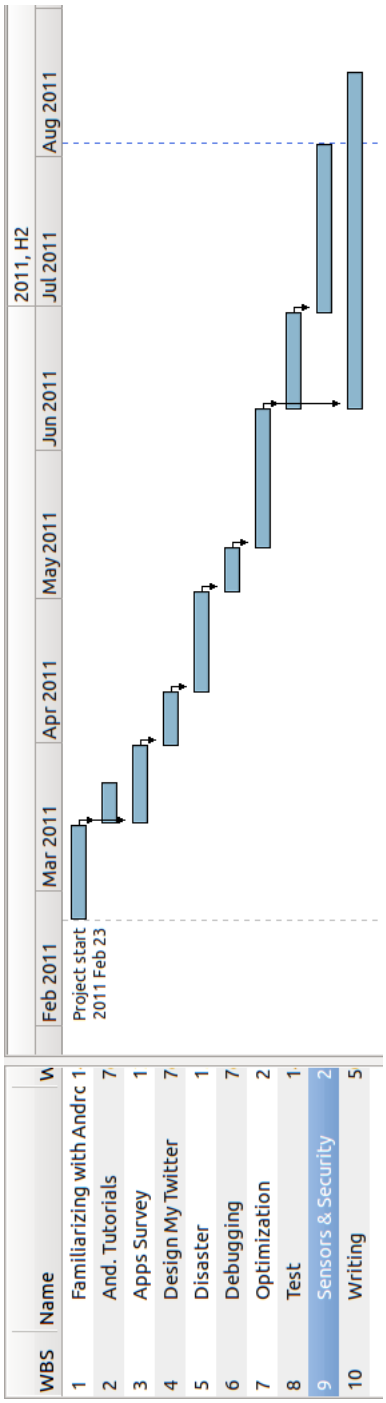


Figure D.1.: Time Schedule

Bibliography

[1] Social universe: http://jess3.com/geosocial-universe-2/.

[2] Tweets per day: http://blog.twitter.com/2010/02/measuring-tweets.html.

[3] Volume of internet traffic in japan : http://www.comscoredatamine.com/2011/03/mobile-internet-traffic-in-japan-spikes-in-response-to-the-tsunami.

[4] Internet in egypt offline: http://bgpmon.net/blog/?p=450.

[5] Android developers: http://developer.android.com.

[6] Phone-os-graphic: http://www.narominded.com/wp-content/uploads/2011/06/phone-os-infographic.jpg.

[7] Podnet project. http://www.podnet.ee.ethz.ch.

[8] Apps survey: http://www.engadget.com/2010/05/06/twitter-for-android-the-best-apps-reviewed/.

[9] Usage statistics: http://developer.android.com/resources/dashboard/platform-versions.html.

[10] Impact on the telecoms network: http://about.ovum.com/app/japanese-earthquake-impact-on-the-telecoms-network/.

[11] Axel Bruns. Tracking crises on twitter: Analysing #qldfloods and #eqnz. Presented at the Emergency Media and Public Affairs Conference, Canberra, 2011.

[12] R. Bruno, M. Conti, and E. Gregori. Mesh networks: commodity multihop ad hoc networks. *Communications Magazine, IEEE*, 2005.

[13] Matthias Grossglauser and David N. C. Tse. Mobility increases the capacity of ad hoc wireless networks. *IEEE/ACM ToN*, 10, 2002.

[14] Bo Han, Pan Hui, V.S. Anil Kumar, Madhav V. Marathe, Guanhong Pei, and Aravind Srinivasan. Cellular traffic offloading through opportunistic communications: a case study. In *Proceedings of the 5th ACM workshop on Challenged networks*, CHANTS '10, pages 31–38, New York, NY, USA, 2010. ACM.

[15] V. Vukadinovic, G. Karlsson. Spectral efficiency of mobility-assisted podcasting in cellular networks. *ACM MobiOpp*, 2010.

[16] Twitters statistics: http://blog.twitter.com/2011/03/numbers.html.

[17] Takeshi Sakaki. Earthquake shakes twitter users: Real-time event detection by social sensors. *ACM WWW*, 2010.

[18] Japan data usage: http://www.telecomasia.net/node/20894.

[19] Japan quake hits cables: http://www.telecomasia.net/node/20792.

[20] J. Sutton, L. Palen, I. Shklovski. Backchannels on the front lines: Emergent use of social media in the 2007 southern california fires. *Proc. ISCRAM*, 2008.

[21] Sarah Vieweg, Amanda L. Hughes, Kate Starbird, Leysia Palen. Microblogging during two natural hazards events: What twitter may contribute to situational awareness. *ACM CHI*, April 2010.

[22] Twitter grows up in aftermath of haiti earthquake: http://designmind.frogdesign.com/blog/twitter-grows-up-in-the-aftermath-of-haiti-earthquake.html.

[23] Speak to tweet: http://googleblog.blogspot.com/2011/01/some-weekend-work-that-will-hopefully.html.

[24] Dtnrg: http://www.dtnrg.org/wiki.

[25] Kevin Fall. A delay-tolerant network architecture for challenged internets. In *ACM SIGCOMM Conference*, pages 27–34.

[26] Vincent Lenders, Martin May, Gunnar Karlsson, and Clemens Wacha. Wireless ad hoc podcasting. *SIGMOBILE Mob. Comput. Commun. Rev.*, 12:65–67, January 2008.

[27] Erik Nordström, Per Gunningberg, and Christian Rohner. A search-based network architecture for mobile devices. Technical Report 2009-003, Department of Information Technology, Uppsala University, January 2009.

[28] Photoshare haggle demo. http://code.google.com/p/haggle/wiki/PhotoShare.

[29] Russell J. Clark, Evan Zasoski, Jon Olson, Mostafa Ammar, and Ellen Zegura. D-book: a mobile social networking application for delay tolerant networks. In *Proceedings of the third ACM workshop on Challenged networks*, CHANTS '08, pages 113–116, New York, NY, USA, 2008. ACM.

[30] Bundle protocol specification: http://www.dtnrg.org/docs/specs/draft-irtf-dtnrg-bundle-spec-03.txt.

[31] Development of android twitter client: http://marakana.com/forums/android/examples/67.html.

[32] Bloa example: http://github.com/brione/brion-learns-oauth.

[33] Sqlite database: http://www.sqlite.org/.

[34] Jtwitter apis: http://www.winterwell.com/software/jtwitter.

[35] Twitter uses oauth: http://blog.twitter.com/2010/08/twitter-applications-and-oauth.html.

[36] Sacha Trifunovic, Bernhard Distl, Dominik Schatzmann, Franck Legendre. Wifi-opp: Ad-hocless opportunistic networking. *CHANTS'11*, September 2011.

[37] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[38] Certicom Research. Standards for efficient cryptography, SEC 1: Elliptic curve cryptography, September 2000. Version 1.0.

[39] Nils Gura, Arun Patel, Arvinderpal Wander, Hans Eberle, and Sheueling Chang Shantz. Comparing elliptic curve cryptography and rsa on 8-bit cpus. In *CHES'04*, pages 119–132, 2004.

[40] Japan phone operators: http://www.businessweek.com/news/2011-03-11/japan-phone-operators-report-poor-services-after-earthquake.html.

[41] A.-K. Pietiläinen, E. Oliver, J. LeBrun, G. Varghese and C. Diot . Mobiclique: middleware for mobile social networking. *WOSN*, 2009.