WiFi Direct Message Flooding API

Distributed Systems – Project Proposal

Alessandro Nicolussi, Joel Busch, Manuel Grossmann, Claude Hähni, Pascal Oberholzer, Jakob Meier
13-942-222, 13-929-559, 13-936-323, 13-929-773, 13-918-115, 13-925-573
anicolus@student.ethz.ch, buschjo@student.ethz.ch, manuelgr@student.ethz.ch, chaehni@student.ethz.ch,
pascalo@student.ethz.ch, jakmeier@student.ethz.ch

ABSTRACT

It's that time of the year again, when all the large festivals and parades happen. It's the time, when all of us give in to one of them and go there, like every year. You go there with a few friends and soon enough you will lose one of them, because the crowd is just too big and too loud. Every one of us knows this situation, right?
It's in this situation when you really need to use your mobile phone, but ending up being annoyed at the absence of ANY reception.
So that's the point where we started...

We want a messaging system to work, even if you can't reach your friends over the internet. We did not just want to program a messaging app for that purpose, but rather go a step further and build an API to provide these functionalities to programmers, so they can develop their own apps with even different approaches than usual.
To demonstrate that our API is usable not just for the messaging system but has more general applicability we also build a second app which aims to deliver an emergency message to a server in the general Internet, using any connection another node in the network may have available.
Our approach to build such a network of nodes in a usable range for mobile devices, is to use WiFi Direct. The API will not forward the messages to a server, it broadcasts them to all nodes. This gives us the highest possibility to get a message to a node without any reception.
So far, so good. But it's not finished yet. There is a lot more stuff needed other than just broadcasting a message through a network of devices. The API should allow a dynamic network structure, which means at any time a node can leave or join the network, it must also buffer the messages to allow reaching nodes which are unreachable for the network at that moment. This leads to a lot of challenging problems with replacement orderings, timeouts and so forth...

1. INTRODUCTION

Our message flooding API can be useful to many future projects that involve several Android devices which should be connected even without a working internet connection. For some applications, the API might simply provide an alternative communication channel that can be used when the device does not have a connection to the internet, but for other applications it can be the core of the communication between several devices.
Two simple example applications will be distributed along with the API as a demo. The first demo is an SOS forwarding app that uses our API to propagate an emergency call between devices which are not connected with the internet, until it reaches a device with a working internet connection that can send the call to a webserver.
Of course the full power of the API will only be visible in more complex systems. In principle, the API will be powerful enough to support a document editor which is synchronized over many users, all without the need of a working internet connection. That could be interesting for a military office outside, but also for a working team that wants to keep working on the same files while traveling together in an airplane.
To demonstrate how the API is used for more complex applications, we will develop a messenger app. The app will support multiple secure chats that users can join.
As the name suggest, the API provides nothing but a message flooding interface, therefore most of the complexity will be in the client's code outside of the API, namely in the client's application. However, the API solves most of the problems of a distributed system and hides them from the client. The features available in the API are:

- **Dynamic local network**: Devices can form a local network and new devices can enter it dynamically.
- **Message flooding**: A device can easily send a message to all other devices in the local network.
- **Message buffering**: A device which loses connection to the other devices will receive all sent messages when it connects to the local network again.
- **Message reordering**: The ordering of messages sent by one device is preserved on the receiver side.

There are already applications and services which, to some degree, do the same. One such example is the FireChat[1] app. FireChat is a proprietary mobile app that builds on a wireless decentralized mesh network to enable smartphones to connect and send messages to each other. The main difference is, that we develop an API instead of one single application. While our API can be used to implement something similar to FireChat (as we will show with our Chat App example) it is capable of handling many more and very different use cases like the aforementioned SOS app.

2. SYSTEM OVERVIEW

2.1 API

The API will offer the following functionalities to clients:

- Initialize network
- Join network
- Broadcast message
- Register receive message listener

In this section, we discuss how we plan to support these functionalities. But before starting with that, we should have a look at what a network is to us and how it is defined.

2.1.1 Definition of a network

Our system is supposed to be fully symmetric, i.e. there is no device (node) in the network with a special task, all nodes execute the same code. In particular, all nodes can send and receive messages at any point in time. New nodes

can join the network by extending it at any node that is already integrated in the network. Furthermore, separately created networks should be able to be merged into one network.
To define our network accurately, let the ordered pair $a(A, B)$ denote an established connection (using WiFi Direct) between node A and B. Each such pair is associated with the timestamp when the connection establishment happened. There will usually be many such establishments with different timestamps that involve the same two nodes, because the nodes are constantly building up and dropping connections to the nodes in reachable distance. We consider the known network of a node C to be the set of all nodes $N_i, (1 \leq i \leq \#$ of nodes) for which holds:
There is a chain of established connections for some n and a given timeout t

$$a_0(N_i, N_{j_0}) \circ a_1(N_{j_0}, N_{j_1}) \circ \ldots \circ a_{n+1}(N_{j_n}, C)$$

such that a_k happened before a_{k+1} for all $k \leq n$ and a_0 is not older than the timeout t allows.
Informally, the network as seen by a given device D consists of all nodes whose signal could reach D within the predefined timeout.

2.1.2 Data structures

To implement the functionalities described at the beginning, we use a few data structures. We will explain them now.

A message sent between two nodes is composite with a Header containing the LC- and the ACK-Table, as well as the content of a number of messages, shown in the figure below.

Message:

Last Contact Table
ACK-Table
Content of Message 1
⋮
Content of Message n

As a second data structure we have the Acknowledgement, seen below, which is just a message with no content.

Acknowledgement:

Last Contact Table
ACK-Table

In the two figures above we showed the form of the messages, which are being sent. In their headers, they contain two tables, namely the Last Contact Table and the ACK-Table. We start by explaining the Last Contact Table.

Each node in the network has a local Last Contact Table. This table has entries in form of (N_i, T_i), where N are nodes in the network and T are the corresponding timestamps. The timestamp represents the time when the node was last present in the network. That means that the timestamp is updated each time the owner node hears from another node and updates its LC-Table with the earlier timestamps.

Last Contact Table:

N_1	T_1
N_2	T_2
⋮	⋮
N_n	T_n

When we filter out nodes from the LC-table which are older than a specified timeout t, then the list of nodes in such a local table corresponds to our definition of the network seen by this node. To build up the next data structure, we will also need this list of nodes considered to be in the known network.

The ACK-Table describes which Receiver nodes got a message from a particular Sender node. The table contains in the first column the Sender nodes and in the first row the Receiver nodes. Each entry in the table (except for the first row and the first column) contains a sequence number of the message from a Sender node, which the Receiver last received.
Important to mention is, since the API is based on a decentralized system, that the table only shows the view seen by the owner of the table at a given time.

ACK-Table:

Sender \ Receiver		N_1	N_2	...	N_n
				...	
N_1				...	
N_2				...	
⋮				⋱	
N_n				...	

We will use this ACK-table to keep track of which messages have definitively reached all nodes, thus can be removed from the local buffer, while other messages should be kept so we can deliver them to the nodes which have possibly not received them, yet.

2.1.3 Implementation

Now that we have discussed the involved data structures and how we define the term network, we can have a look at the actual implementation plan. Devices (nodes) establish connections over WiFi Direct and the protocol we use to communicate will be TCP.
To **initialize** the network, we have to create an ACK- and a LC-table with only our own entry. Then we simply contact other nodes and perform a merge of the tables whenever we encounter tables with different entries.
Joining a network for a single node is basically just initializing the network as if it was new. However, we might want to ask around for old messages that the other nodes have still stored in their local buffers.
Broadcasting a message is done by adding the message to the local buffer and then invoking the send mechanism. The send mechanism goes through the local buffer and determines the messages which have not reached its neighbours. Neighbours are all those nodes which are currently visible. If there are neighbours which do not have all the locally buffered messages, then we send them all missing messages.
The **message listener** provided by the client will be called whenever a new message arrived. Right after calling the listener in a new thread, we can mark in the ACK-Table that we received that message.
To allow above functionalities, we have to implement a way of merging networks, i.e. merging the LC- and ACK-tables of different networks. Further, we have to build up WiFi Direct connections with all nodes in our reach. We can build up the LC-table in a way that we know the MAC addresses of all nodes, therefore we will know which visible devices are part of our network. With those we will build up a connection whenever we receive data that, according to our ACK-table, has not reached this node. If the entry of a node in our LC-table is getting older than a predefined threshold, then we also try to establish the connection again, to keep the information about the network up to date.

Devices which are not part of our network have to be considered as well, they could be in a network that could be merged with ours. Therefore we keep a list of devices that are not part of our network, too. For each such device, we store the timestamp when we attempted to merge the last time. When this entry is getting old and the device is still visible, we retry.

2.1.4 Seperating the network from the application

The API that we develop can potentially be used by many applications. Useres might want to use several of them at the same time. Although it is feasible to have a seperate network for each application, we decided to follow a different approach. Applications should be able to share the network with other applications, improving the connectivity between nodes. On the other hand, we definitely need a way to seperate networks from each other, to have some control over the network size.
The solution we want to apply is giving the power to choose a network to the end-user. Application developers can of course advise the users to join a specific network, so that all their users are connected, but eventually the user can decide with whom he or she wants to share a network.
Technically, we achieve that by running our API in its own process and providing a user interface to change the network settings. Client applications can then communicate with our API using interprocess communication (IPC). The API will send messages between all nodes in the network, even if the different nodes are using different applications. Applications will only be notified about messages that have been sent by the same application, a simple application tag associated with the messages will allow us to decide this within the API efficiently.

2.1.5 Target client applications

Even though our API can serve as backend to a variety of different application, some of our design choices are based on assumptions on how it will be used.
First, we assume the network to consist a relatively small number of nodes, typically not more than 20 nodes. We will support more nodes, but the performance might be very poor and we will *not* test our code for good scalability to hundreds of nodes.
Second, a user can only be in one single network at the time. The limitation is not on number of applications, but on groups of users, who decided on a common network. If someone uses our API with two independent such groups who are not willing to share a single network, the user will not be able to connect to both networks at the same time. But of course the network can be changed manually as often as needed.

2.2 Emergency App

The main idea of this application is to provide emergency services even if a cellular connection cannot be directly established.
Users have to enter some personal data (name, address, birth date, insurance number (optional), allergies (optional) etc.) when launching the App.
Whenever a user gets into an unpleasant situation, he/she can set off an emergency message via the App (Graphic - Button Press).
The message contains the user's personal data, as well as his/her GPS coordinates at the time the emergency message was successfully sent.
Emergency App takes care of forwarding the message to a PoH (Point of Help). If cellular network is available, the emergency message is set off directly.
What if there is no direct connection? As soon as another user of the App is reachable via the WiFi Direct API, the emergency message is sent to that user who immediately gets notified that someone needs help. In case the new user is capable of a network connection, the message is sent to a PoH via his/her network connection. If not, the message is forwarded to another reachable user of the app. The message propagates across the growing chain of WiFi Direct connections and is flooded across the resulting network until a direct connection to a PoH can be established (SMS, TCP segment). The PoH then acks the message and the ACK is propagated along the network of users to stop the flooding and tell the victim that help is on the way.
Moreover, users on the WiFi direct chain get an estimation of the cardinal direction of where the emergency message was set off relative to their position in order to administer first aid. However, if location services are not available to the victim (i.e. due to being stuck in a tunnel or cave), the first node on the emergency chain which can determine its GPS location puts it onto the message. This gives a reasonable approximation of the victim's location.

2.3 Chat App

One of the primary uses of any electronic network is private communication. The chat app is therefore intended to let any two people or, more precisely, any two devices on the same network communicate privately, despite all underlying messages being distributed to any and all reachable nodes.
It is inherently impossible to know how a message needs to be routed in the underlying topology because it is dynamic. That is why we can not adapt the network to the use case of the app, but instead use cryptography to build the private channels on top of its broadcasting. Privacy is established using public key cryptography. Messages are encrypted by the sender using the public key of the receiver to ensure only they can decrypt them. Each user generates their own key pair and can then share their public key via QR code. The chain of trust works in one of two ways, either, users wishing to communicate privately later scan each others public keys directly, or a chain of trusted third parties relay the public keys.
Since the underlying network has no concept of a receiver, and only relays messages on a best-effort basis it is up to the chat application to report back a successful message delivery. The acknowledgements too have to be encrypted, otherwise an eavesdropper might observe communication behaviour and reach conclusions based on timing and frequency. However since the acknowledgements may fail often or suffer wildly inconsistent travel times no automatic retransmission of messages is attempted. The decision is relegated to the user.
A chat app message will consist of:

- a magic value that signals successful decryption
- an acknowledgement for the last seen sequence number of the chat partner
- a time-stamp
- a sequence number
- the senders identifier
- the payload text

It will be limited in size such that it fits into an underlying network message, longer user texts will be broken into parts if need be. A simple acknowledgement message will only contain the first two fields of the aforementioned. The whole

message with all those headers is encrypted. This implies that each message that reaches the app must be decrypted using the private key and most will not yield the magic value, because the device is not the intended receiver, however that is feasible for our small networks and protects meta-data about communications.
When first starting the app the user has to enter his name and generate his public and private key. After generating the key the user is able to scan public keys from other members or provide his own public key for scanning from the overflow menu of the thus-far empty main view. Once some keys have been scanned a list of chats starts to populate in the apps main view. Pressing on a chat in the chat-list opens a chat to write and read messages. From the overflow menu it will be possible to share the current chat partners key further. Thus the chat list doubles as the address book.
A QR code will contain:

- a magic value that signals the scanned code belongs to the app
- the chat partners name or pseudonym
- the chat partners public key

Scanning a QR code will open a new chat and remind the user that the other can only answer if they also possess your public key.
A chat view is similar to other popular messaging apps, in that the messages can be attributed due to the presentation using either colour or positioning. The time-stamp included in the messages are displayed as the time of sending, precision of the chat partners clock can not be guaranteed however, so the times are only as good as the chat partner can vouch for his clocks precision. The acknowledgement status of each message is made visible using colour or iconography.

3. REQUIREMENTS

Several choices have to be made that limit the reach of our application, in order to keep the project simple enough for the given time frame. Perhaps the choice that limits us most is using the Wi-Fi Peer-to-Peer API in Android. It constrains us to devices that have at least Android 4 (API level 14) installed and that have hardware capable of Wi-Fi Direct communication.[3]
For reading and generating QR codes we will use the ZXing project[4]. We will prompt user to install the ZXing Barcode Scanner app if not already available. It's installation size is in the neighbourhood of 1MB depending on the device, so the size is not a problem. The app requires API level 16, access to the Google Play Store and of course a device with a camera. However we provide an alternative, if somewhat arduous, method of copying the public keys by typing them in manually.
Beyond that we will use only standard Java and Android libraries so no further limitations apply to the system software.
We will develop three apps, one being a wrapper around the core networking service, with a simple interface to enter some configuration details, the others being the emergency app and the chat app. They are going to use an API to access network functionality from the first. Each app individually will be rather lightweight so storage concerns should be fairly insignificant since we don't include much audiovisual content in any of these components. In case the users wish to use one app with one group of people and the other with another, they will have to reconfigure their network association in the wrapper app repeatedly.
We are depending on users to judge the performance of the network on their respective devices and make sure they keep their networks small enough before scaling issues make it unusable.

4. WORK PACKAGES

4.1 API

1. Define the IPC interface of our API and hand a AIDL[2] file to the other group members
2. Establish Peer-to-Peer connection with WiFi Direct
3. Build data structures for LC-Table and ACK-Table
4. Implement network initialization
5. Implement functions to update tables
6. Build message and parse message
7. Build data structure for local buffer
8. Implement message sending (broadcast)
9. Implement message receiving (with message listener)
10. Write code for merging two existing networks
11. Implement buffer entry replacement strategy
12. Remove old nodes from network according to a timeout specified by the client
13. Correct reconnection
14. Request all buffered messages from a node

4.2 Emergency App

1. Main Activity with "request help" button. Button is only clickable if personal information is entered and location services are turned on. On button click the user can select what kind of emergency case it is.
2. Settings Activity which stores personal information such as name, insurance numbers, allergies, etc.
3. Notification Activity which shows a relayed emergency request on the users phone including walk directions to find the requester.
4. A webserver which distributes the request to the specific emergency services in charge.

4.3 Chat App

1. Create a MainActivity with clickable list of chats ordered by most recent activity. Each chat should display how many messages are unread.
2. Add an overflow menu with "Preferences", "Show Key", "Add Chat", "Go Offline" and "Open Network Configuration" entries.
3. Implement a service that handles message state, address book state, receiving messages including decryption and notification handling. It is to be started on app start if not running.
4. Store chat content, address book and own keys in separate files, when the service is shut down.
5. Create a ChatActivity that holds the chat window, with message list left and right aligned, depending on sender, ordered descending in age.

6. Add an overflow menu with ”Show Key” and ”Forget User”.

7. Using the ZXing Barcode Scanner allow for displaying keys and scanning public keys.

8. Create an activity to generate a public-private key pair with java.crypto and to enter the user-name.

9. Create a preference activity with two options to toggle sound and vibration for notification and an option to generate a new key.

5. MILESTONES

First of all the public function signatures of our API are defined and handed to the other group members that they can start with the Emergency App and the Chat App. Then the API team works at the remaining work packages and the other group members can start with their work on the emergency app and the chat app. The emergency app team will partially support the API team until the work packages 1 to 9 are met.

Before the emergency app and the chat app can be tested the API has to be finished because the two apps rely on the message forwarding of the API.

Schedule:

Date:	Subject to finish:	Responsible:
20 Nov	Function overview API	Manuel, Jakob
24 Nov	Emergency App UI complete	Alessandro, Claude
25 Nov	Chat app up to WP3 complete	Joel, Pascal
4 Dec	Chat app up to WP5 complete	Joel, Pascal
4 Dec	API: Basic send/recv. (up to WP9)	Manuel, Jakob
10 Dec	Observable API behavior is stable	Manuel, Jakob
11 Dec	Chat app complete for testing	Joel, Pascal
11 Dec	Emergency App: able to set off and display requests	Alessandro, Claude
12 Dec	Presentation slides	all
14 Dec	Emergency App: Webservice for distribution of requests running	Alessandro, Claude
18 Dec	All tasks complete	all

6. REFERENCES

[1] Open Garden, FireChat application. http://www.opengarden.com/firechat.html.

[2] Google. Android interface definition language. https://developer.android.com/guide/components/aidl.html.

[3] Google. Wi-Fi Peer-to-Peer API Guide. https://developer.android.com/guide/topics/connectivity/wifip2p.html.

[4] S. Owen, D. Switkin, and other ZXing contributors. ZXing Project Page. https://github.com/zxing/zxing/.