WiFi Direct Message Flooding API

Distributed Systems – Project Proposal

Student One, Student Two, Student Three, Student One, Pascal Oberholzer, Jakob Meier
ETH ID-1 XX-XXX-XXX, ETH ID-2 XX-XXX-XXX, ETH ID-3 XX-XXX-XXX, 13-929-773, 13-918-115, 13-925-573
one@student.ethz.ch, two@student.ethz.ch, three@student.ethz.ch, chaehni@student.ethz.ch, pascalo@student.ethz.ch, jakmeier@student.ethz.ch

ABSTRACT

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.
A simple example application will be distributed along with the API as a demo. The 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 systems 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.

2. SYSTEM OVERVIEW

2.1 API

The API will offer the following functionalities to clients:

- Initialize network
- Join network
- Leave 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 message at any point in time new nodes can join the network by extending it at any node that is already integrated in the network.
So, what is a network in our case? Let the pair $a(A, B)$ denote an established connection between node A and B. 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 the device within the predefined timeout.

2.1.2 Data structures

To implement the functionalities described at the beginning, we establish connections over WiFi Direct and use a few data structures. Since we use these data structures later in our overview, 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 the message, shown in the figure below.

Message:

Last Contact Table
ACK-Table
Content

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 updataed each time the owner node hears from another node or it updates it's own LC-Table with one it got from another node.

Last Contact Table:

N_1	T_1
N_2	T_2
$\vdots$	$\vdots$
N_n	T_n

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:

Receiver

Sender

	N_1	N_2	$\dots$	N_n
N_1			$\dots$	
N_2			$\dots$	
$\vdots$			$\ddots$	
N_n			$\dots$	

2.1.3 Implementation

Now that we have discussed the involved data structures and how we define network, we can have a look at the actual implementation plan.
To **initialize** the network, we have to create a 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 more 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.
Leaving a network is not reall necessary for correctness, but it can make the whole network more efficient if nodes notify the network when they are not interested in the messages anymore. (**OPEN QUESTION**: Special signal that tells all nodes to delete that node? Or, setting the seq number to ∞ and only removing it after some time? Other ideas?)
Broadcasting a message (is also an **OPEN QUESTION** but here is my suggestion,) 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 it 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.
TODO: merging

2.1.4 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 consists of a relativly 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 optimize our code for good scalability to hundreds of nodes.
TODO: Second, (What ever we decide about the many networks on a single device problem)

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

The Chat App ensures end to end encrypted messages via peer-to-peer connection through the flooding API. Encrypting and Decrypting messages is done public key cryptography. The keys are generated by the user and shared by QR codes that have to be scanned from an other user which acts as a chat partner.
If the receiver's network is not connected to the sender's network the messages are buffered and will be sent to the receiver later when the receiver's and the sender's network are connected. The receiver is able to get as many messages as are stored in the buffer of the API.
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. After scanning a public key the App reminds to scan the public key of the other chat partner. If only one chat partner has scanned the

public key it will only be possible to send messages in one direction (one is acting always as a sender, the chat partner is acting always as a receiver).
Pressing on a chat in the chat-list opens a chat to write and read messages. The messages are simple text messages with a maximum length which is determined by the API.

3. REQUIREMENTS

Joel
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.[1]

4. WORK PACKAGES

4.1 API

1. Define all public function signatures of our API and hand it 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 with two nodes
5. Write code for nodes to join an existing network
6. Implement functions to update tables
7. Build message and parse message
8. Implement message sending (broadcast)
9. Implement message receiving (with message listener)
10. Build data structure for local buffer
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
15. Unsubscribing nodes from the network

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 activity. Each chat should display how many messages are unread.
2. Add a overflow menu with "Preferences", "Show Key", "Add Chat", "Go Offline" buttons.
3. Implement a service that handles message state, address book state, receiving messages including decryption, notification to be started which starts on app start if not running.
4. Chat, address book and own keys have to be stored in separate files, when the service is shut down.
5. ChatActivity: Chat window, with message list left and right aligned, depending on sender, ordered descending in age
6. Add an activity to generate a public-private key pair with java.crypto.
7. Using ZXing library make two activities, one for displaying keys and one for scanning them.
8. Add and activity for initial key generation and name entry.
9. In preference menu add two options to enable sound and vibration for notification. Furthermore add 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.

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
14 Dec	Emergency App: Webservice for distribution of requests running	Alessandro, Claude
18 Dec	all tasks complete	all

6. REFERENCES

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