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, ETH ID-1 XX-XXX-XXX,
13-918-115, 13-925-573
one@student.ethz.ch, two@student.ethz.ch, three@student.ethz.ch, one@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 alternetive 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 travelling 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 copmplexitiy 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 ditributed 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. 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 conection between node A to node B. We consider the known network of a node C to be the set of all nodes $N_i, (1 \leq i \leq \#\text{ of nodes})$ for which holds:
There is a chain of established connections for some n and a given timeout t

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

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 consits of all nodes whose signal could reach the device within the predefined timeout.

Last Contact Table:

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

ACK-Table:

Sender \ Receiver	N_1	N_2	$\ldots$	N_n
N_1			$\ldots$	
N_2			$\ldots$	
$\vdots$			$\ddots$	
N_n			$\ldots$	

Message:

Last Contact Table
ACK-Table
Content

Acknowledgement:

Last Contact Table
ACK-Table

2.2 Emergency App

Claude, Alessandro
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 unpleasent 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

Joel, Pascal 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 the receiver.
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.
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. Upon scanning a new chat is displayed in the chat-list and a reminder appears to scan the public key of the chat partner.
Pressing on a chat in the chat-list opens a chat to write and read messages.

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.[**?**]

4. WORK PACKAGES

4.1 API

Jakob, Manuel

4.2 Emergency App

Claude, Alessandro

4.3 Chat App

Joel, Pascal

1. MainActivity: Clickable list of chats ordered by activity with names and unread message counter overflow menu with "Preferences", "Show Key", "Add Chat", "Go Offline"
2. Storing chats, address book and own keys in files when service is shut down
3. ChatActivity: Chat window, with message list left and right aligned, depending on sender, ordered descending in age
4. Using ZXing library make two activities, one for displaying keys and one for scanning them
5. Preferences, for sound and vibration and new key generation
6. Service that handles message state, address book state, receiving messages including decryption, notification to be started on app start
7. Generating public-private key pair with javax.crypto
8. Activity for initial key generation and name entry

5. MILESTONES

Pascal