FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Spontaneous Social Networks

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Michael Hanzl
Matrikelnummer 0525742

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Projektass.(FWF) Dipl.-Ing. Dr.techn. Peter Rautek

Wien, 01.07.2011

(Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 01.07.2011 _____

(Unterschrift Verfasser)

Abstract

Social networks are a well researched field. There is extensive literature covering this topic. They are nowadays used for a wide variety of applications such as business, education, medicine and dating. These applications contribute to the high popularity of social networks. The underlying concept of all social networking applications is a graph, that consists of nodes representing people and edges representing social relations. This social graph is commonly very rigid in semantics and tedious to manipulate. Two users can add an edge only by mutually accepting the relation. Typically this graph manipulation needs manual intervention.

In contrast, Spontaneous Social Networks (SSNs) rely on much more flexible graphs that are manipulated with minimal manual intervention. Edges are automatically generated and deleted in real time by hardware devices such as smartphones and desktop PCs. Applications operate on top of SSNs. The application defines the purpose and properties of the SSN and provides the users an interface to a SSN.

The goal of this diploma thesis is the design of a standard and an implementation of a Spontaneous Social Networking (SSN) framework. The capabilities of the framework are demonstrated by a number of example applications. The Extensible Messaging and Presence Protocol (XMPP) and the Publish-Subscribe (PubSub) paradigm are used for communication in SSNs. The properties and discovery approaches of SSNs are defined. The implementation is evaluated in terms of usability with the help of the System Usability Scale (SUS). Furthermore, the suitability of XMPP and the scalability and privacy of SSNs and the reference implementation are discussed.

Kurzfassung

Soziale Netzwerke sind ein gut erforschtes Gebiet, mit entsprechend viel wissenschaftlicher Literatur. Soziale Netzwerke werden heutzutage für viele verschiedene Anwendungsfelder, wie z.B. im geschäftlichen Umfeld, für den Unterricht und die Lehre, in der Medizin und um andere Personen kennen zu lernen, verwendet. Diese Anwendungsgebiete machen Soziale Netzwerke sehr populär. Das zugrunde liegende Konzept eines Sozialen Netzwerks ist ein Graph, der aus Knoten und Kanten besteht. Personen werden als Knoten dargestellt und die sozialen Beziehungen zwischen diesen Personen als Kanten. Der soziale Graph ist semantisch sehr starr und mühsam zu verändern. Eine Verbindung zwischen zwei Personen muss normalerweise von beiden Personen akzeptiert werden und erfordert ein manuelles Eingreifen der Benutzer.

Spontane Soziale Netzwerke - Spontaneous Social Networks (SSNs) - verwenden im Gegensatz zu Sozialen Netzwerken viel flexiblere Graphen, bei denen das Eingreifen des Benutzers auf ein Minimum reduziert wird. Kanten werden in Echtzeit von Geräten, wie z.B. Smartphones und Desktop PCs, erzeugt und gelöscht. Auf den SSNs operieren Applikationen, die den Zweck und die Eigenschaften des Spontanen Sozialen Netzwerkes festlegen. Außerdem bieten sie dem Benutzer eine Schnittstelle zur Benutzung von SSNs.

Das Ziel dieser Diplomarbeit ist das Design eines Standards und die Implementierung eines Frameworks für SSNs. Etliche Beispielapplikationen demonstrieren die Funktionen und Fähigkeiten des Frameworks. Für die Kommunikation in SSNs wird das Extensible Messaging and Presence Protocol (XMPP) und das Publish-Subscribe Modell verwendet. Die Eigenschaften und Ansätze für das Auffinden von SSNs werden behandelt. Die Implementierung wird mit Hilfe des System Usability Scale (SUS) in Bezug auf die Brauchbarkeit bzw. Benutzerfreundlichkeit evaluiert. Außerdem wird die Tauglichkeit von XMPP, die Skalierbarkeit sowie der Datenschutz in SSNs und der Beispielimplementierung diskutiert.

Acknowledgements

First I want to thank Peter Rautek, my supervisor, for his great support. He supported me with new ideas and feedback, gave me valuable tips, proofread this thesis and solved administrative problems quickly and uncomplicated. Next I want to thank Eduard Gröller, our beloved master, for his advises and proofreading this thesis.

Furthermore, I want to thank the Institute of Computer Graphics and Algorithms for lending me an Android smartphone during my diploma thesis and for providing me a server to host the XMPP server. Moreover, I thank Andreas Maierhofer who administrates the server and gave me tips during the setup of the XMPP server.

A big thanks goes to the participants of the two performed usability (SUS) tests. They gave me great feedback and tips after the usability tests. I also would like to thank John Brooke for developing the System Usability Scale (SUS).

A special thanks goes to my parents Ludwig and Maria for supporting me and financing my study.

Contents

List of Figures

List of Tables

List of Abbreviations

AIDL	Android Interface Definition Language
API	Application Programming Interface
APK	Android Package
CAPTCHA	Completely Automated Public Turing Test to Tell Computers and Humans Apart
CPU	Central Processing Unit
EPL	Eclipse Public License
FFT	Fast Fourier Transform
GPS	Global Positioning System
ID	Identifier
IDL	Interface Definition Language
IESG	Internet Engineering Steering Group
IETF	Internet Engineering Task Force
IPC	Inter Process Communication
IQ	Info/Query
IT	Information Technology
JID	Jabber Identifier
JSF	Jabber Software Foundation
MANET	Mobile ad-hoc Network
OS	Operating System
P2P	Peer to Peer
PC	Personal Computer
PCM	Pulse Code Modulation
PEP	Personal Eventing Protocol

PubSub	Publish-Subscribe
QR	Quick Response
RAM	Random-Access Memory
RFC	Request for Comments
RPC	Remote Procedure Call
RSM	Result Set Management
SDK	Software Development Kit
SQL	Structured Query Language
SSN	Spontaneous Social Network
SUS	System Usability Scale
TCP	Transmission Control Protocol
TTL	Time to Live
UDP	User Datagram Protocol
UI	User Interface
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
WLAN	Wireless Local Area Network
XEP	XMPP Extension Protocol
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
XSF	XMPP Standards Foundation

Chapter 1

Introduction

1.1 Motivation

"People have always been social creatures; our ability to work together in groups, creating value that is greater than the sum of its parts, is one of our greatest assets." [WM08]. This is one of the main reasons why online social networks are so popular. The internet and the emerged IT technologies simplify the formation of social networks, but they also exist without them. People think of social networking sites such as Facebook, MySpace and Twitter. These are lethargic, because each relationship has to be mutually accepted. Furthermore, the relationships of one person tend to stay constant after some time and social networking sites are usually not very diversified.

In contrast to traditional social networking sites, Spontaneous Social Networks (SSN) are more agile. While traditional social networking sites pretend that there is one social network for each person, SSNs try to model the ubiquitous availability of social networks. They are small, short-lived and implicitly defined social networks. Relationships are created in realtime, and with minimal manual intervention.

Applications operate on top of SSNs. The application defines the purpose and some properties of the SSN and provides the users an interface to a SSN. Nowadays, communication among different groups is often parallel, flexible and short-lived. For instance, a person schedules an event with her/his colleagues, chats with people that are interested in tennis and shares photos with people that plan a trip to Canada simultaneously. There is an own SSN for each of these activities. The SSN members do not notice that other members use several SSNs simultaneously. Mobile devices such as smartphones and tablet PCs offer great opportunities for building and maintaining social networks. SSNs benefit from these opportunities such as the ability to retrieve the accurate user's location, a permanent internet connection and sensor data.

Figure 1.1 illustrates SSNs. All SSNs that are joined by user A are shown. For instance, user A is located at a pub and posts videos and text messages on a shoutwall, schedules a date for the next journey with her/his friends and receives notifications from her/his boss (user F) about the work that user A must complete tomorrow. Each of the three shown SSNs defines a different social relation among its members. The members of SSN 1 schedule a date collaboratively for a journey. The members of SSN 2 are located at a pub and post YouTube videos and text messages on a shoutwall. The members of SSN 3 are co-workers and interested in sending/receiving notifications. User B, who schedules a date for the journey with user A, is also located at the pub and uses the shoutwall.

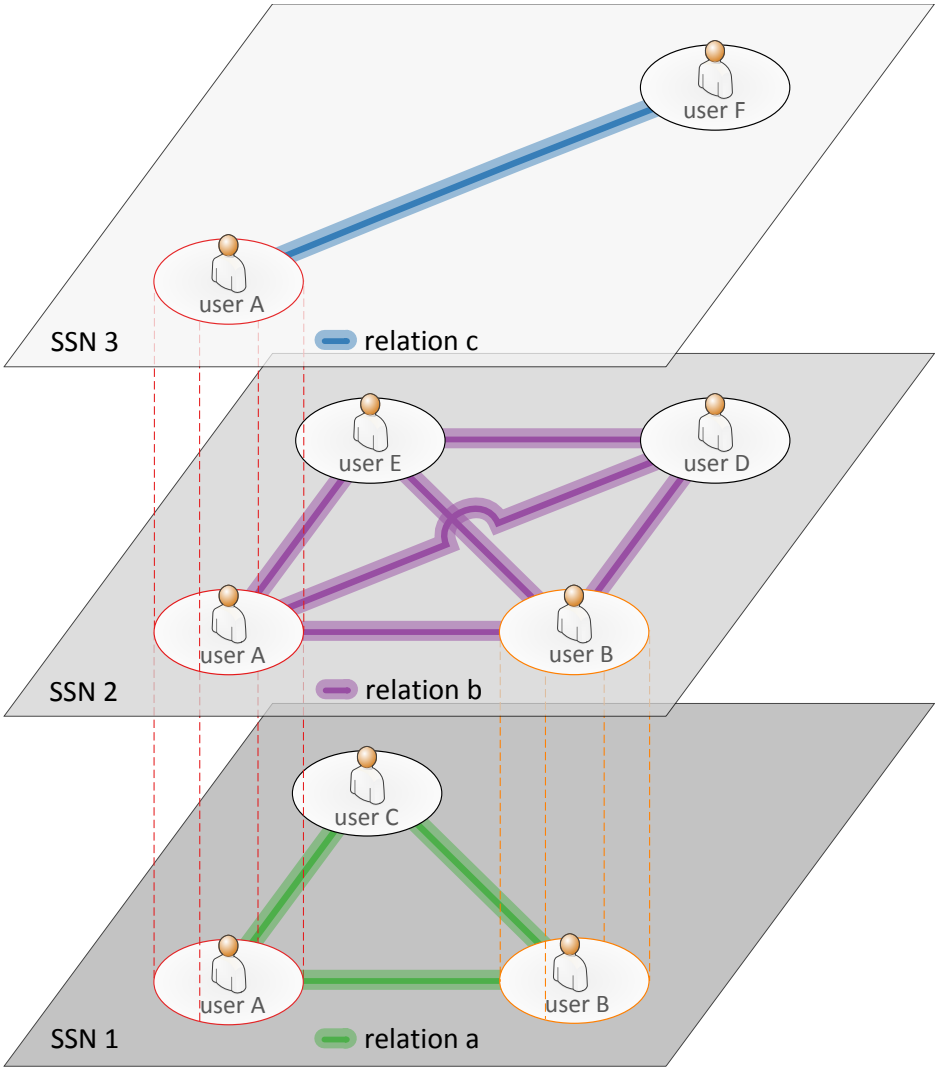


Figure 1.1: Illustration of SSNs: All SSNs that are joined by user A are shown. Each of the three shown SSNs defines a different social relation among its members. User B has joined SSN 2 and SSN 3.

There is a vast number of scenarios where SSNs are useful. Some examples are:

- Several people are participating in an event. With the help of an SSN they are able to exchange text messages, contact information, files and media data with the people around them.
- A study/work group wants to schedule the next study / business meeting. With the help of an SSN they are able to schedule a date.
- A person does not know what (s)he should eat at lunch. An SSN enables the person to contact her/his friends and choose a meal by collaboration (i.e., each friend is able to suggest a meal) or by voting for predefined meals.
- A group has to create a document, a drawing or plan a trip collaboratively. An SSN can be used for these issues.

- A group leader wants to notify her/his followers of news or appointments. Therefore, the group leader publishes notifications on an SSN.
- A group of people wants to track each other. The user's locations are shared with the help of an SSN.
- A company's server should keep track of the employee's smartphones. Therefore, the mobile devices send the WLAN signal strength, mobile network signal strength, CPU temperature, CPU load, noise level, location and velocity to an SSN.
- Strangers are interested in finding new contacts. An SSN exchanges the profiles and suggests new contacts based on similarity matching of the profiles.
- A number of users want to control a shared resource (e.g., shoutwall, playlist of videos or music). An SSN enables the users to send the messages/data to the shared resource.
- SSNs can be used for multiplayer games such as poker, capture the flag or other entertainment applications. The game state (e.g., moves, poker hand) is distributed with the help of an SSN.

These scenarios raise a lot of requirements and problems. For instance, all presented scenarios might require access control to an SSN to protect the messages and data. How are people assigned to a specific SSN?

The goal of this diploma thesis is the design and implementation of a generic Spontaneous Social Networking framework. The framework enables users to create and manage SSNs. Three simple example applications are implemented that act on top of SSNs. These example applications demonstrate the capabilities of the framework. Furthermore, this thesis should answer the following research questions:

- Is such a system useful or are people confused when using SSNs?
- Is the Extensible Messaging and Presence Protocol (XMPP) and the publish-subscribe paradigm suitable to realize SSNs?

To answer the first question we perform a usability test (System Usability Scale (SUS)) with a small group of students. The second question is answered by implementing an SSN framework and three example applications using XMPP and the publish-subscribe paradigm.

1.2 Thesis Structure

The remainder of this diploma thesis is structured as follows: Chapter 2 presents the related work. The related work are past or ongoing research projects that have a quite similar idea or goal as this thesis. An evaluation and analysis of the related work conclude Chapter 2.

Chapter 3 presents the fundamentals that are necessary to understand our implementation of SSNs and enables the interested readers to have a deeper insight into the discussed technologies. First, an overview of social networks including definitions, history and a short analysis of current social networking sites is given. Next, the mobile platform Android is discussed. After a short overview including history, spreading, competitors and features, the architecture, the basics of Android and the interprocess communication

are presented. The last two sections of Chapter 3 deal with the Extensible Messaging and Presence Protocol (XMPP) and its extensions. The penultimate chapter presents the architecture and basics of XMPP. The last section describes important XMPP extensions that are used in SSNs.

Chapter 4 describes the SSN concept in detail and the mapping to XMPP. The presentation of the message exchange, architecture and the discovery approaches are the main parts of this chapter. The authentication and privacy section presents the access models and solutions for common privacy issues. Furthermore, the user management and the buddy list are described.

Chapter 5 describes important parts of the implementation and discusses implementation issues. First, the message exchange is described. Then the implementation of discovery approaches via visual codes and via tones are described in detail. Next, an assortment of implementation issues are mentioned. A comparison of available XMPP server implementations, the decision for the XMPP server and a discussion of existing XMPP libraries conclude this chapter.

Chapter 6 presents the implemented example applications. First, a description and meaningful examples for the usage are given. Then the features and implementation issues are described.

Chapter 7 evaluates SSNs, the SSN framework and the example applications. First, the evaluation in terms of usability based on the System Usability Scale (SUS) is presented. Then the scalability of the approach, the suitability of XMPP and the privacy are discussed.

Chapter 8 summarizes this thesis. The conclusions are presented in Chapter 9. The appendix includes the stanzas during the initialization of an XMPP connection and SASL authentication, a message exchange in an SSN, the SSN API, the XML schema of defined XML pieces and the changes to the XMPP client library Asmack and the XMPP server ejabberd.

Chapter 2

Related Work

2.1 SCOPE

SCOPE "is a prototype for spontaneous P2P social networking" [MAMC10]. Spontaneous P2P social networking enables users to communicate, share experiences and comment them in situations and locations such as events, bars and schools. It works on mobile devices in a local area using WLAN (IEEE 802.11) ad hoc mode. SCOPE enables users to share content and it offers a news feed of updated content and events. The system consists of two types of nodes: Super-nodes and client nodes. Super-nodes are devices with high processing and storage power. Client nodes connect to super-nodes to use the system by calling methods on the super-nodes via XML-RPC. A distributed hash table is used for the messaging system. It runs on the super-nodes. All communication is interpreted as put, get and remove operations. As in ordinary hash tables, a value (e.g., shared content) is identified by its key. The key is obtained by applying a hash function on the value. The key indicates the super-node where the content is stored, because each super-node is responsible for a certain range of keys. Beside the mentioned features, SCOPE offers services such as voice over IP, instant messaging, video telephony, v-card sharing, link sharing, searching people and content. Furthermore, it enables users to develop new services. A list defines which nodes support a specific service. With the help of this list a node that offers a particular service is able to join an existing service or create a new service. Figure 2.1 shows the architecture of SCOPE. [MAMC10]

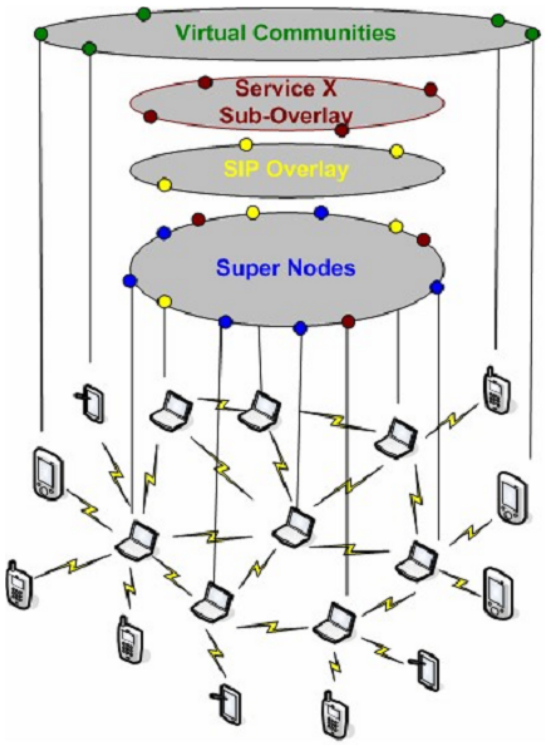


Figure 2.1: SCOPE architecture [MAMC10]: In a spontaneous P2P social network (Virtual Communities) the super-nodes (blue, reddish brown and yellow dots) store and provide the required data, route messages to other super-nodes and provide the services. All other clients (green dots) connect to those super-nodes to use the system. The super-nodes are able to offer predefined services (e.g., SIP proxy - yellow dots) or any other service (Service X Sub-Overlay - reddish brown dots).

2.2 Mobilis

Mobilis is "a service-oriented approach for the development of collaborative applications on top of social networks" [SSS10]. It supports heterogeneous mobile platforms such as iPhone and Android, provides connectors to existing social networks and web services, uses open standards, is extensible, secure and maintains the user's privacy. XMPP, its extensions and custom extensions are used to realize communication in Mobilis. Therefore, a standard XMPP server without modifications is used. Mobilis offers a lot of services that provide the functionality such as collaboration, location and media sharing. So called broker services are introduced for reusability reasons. Mobilis applications connect to the broker services. They bundle functionality provided by the Mobilis services. Broker services are realized by XMPP client bots that are addressed with the help of the resource part of the JID. That is, all bots have the same node and domain identifier. This approach enables applications to use the XMPP service discovery extension [XSF08b] (`disco#items`) to retrieve a list of all available broker services. Figure 2.2 displays the Mobilis service architecture. A plenty of applications are implemented that use the Mobilis platform. Some of these are MobilisMedia, Mobilis Map Draw, MobilisGuide and MobilisBuddy. MobilisMedia is an application for image and video sharing. Each published media is associated with the location, time and user and is stored in a global database. However, it is also possible to control the access to a media file by leaving the file on the creator's mobile device. The media can also be shared via social networks like Facebook. There is a location-based view of all media. Furthermore, media can be filtered by location, time, or user. MobilisMedia uses a lot of services such as the media sharing, media content, social network integration and location service. In the Mobilis Map Draw application users are able to collaboratively draw on a map to for instance create a tourist trip. MobilisGuide is a tourist guide that enables a closed group to share their location and created media to each other. MobilisBuddy alerts users when Facebook friends are near to each other and provides a chat functionality. [SSS10]

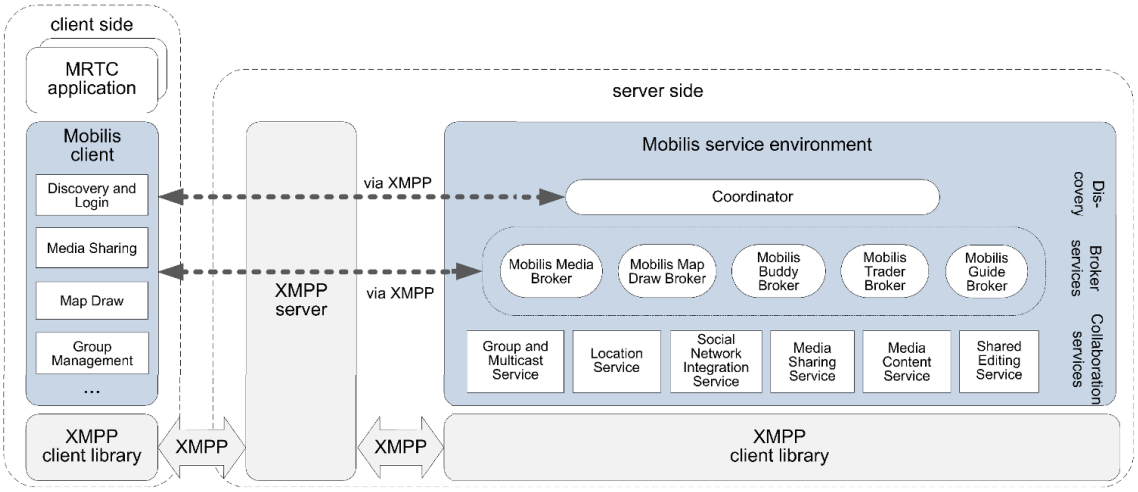


Figure 2.2: Mobilis architecture [SSS10]: It consists of an XMPP server and XMPP client bots on the server side. The client bots provide the broker services and the collaboration services, which are used by the broker services, and provide the functionality for the MRTC applications. The coordinator is used for discovering the broker services. On the client side there are one or several MRTC applications, the Mobilis client for communication with the Mobilis service environment and the XMPP client library to enable communication over XMPP.

2.3 MobiSoC

MobiSoC [GKBB09] is a middleware that enables developers to implement mobile social computing applications (MSCAs). It aims to improve social interactions in physical communities and with unknown people in the physical proximity. It captures, manages and shares the social state of a community. The social state consists of the people's profiles, places and relationships between people and people to places. For instance, MSCAs are able to answer questions if friends / colleagues are located in a certain place, if anybody would like to do a certain activity or where the people doing a certain activity are located. MSCAs can be subclassed into people-centric and place-centric applications. People-centric applications use social and location information including their history to find contacts. Place-centric applications ask people that are at a place of interest and might be able to answer a certain question. MobiSoC is a centralized approach using SOAP over the internet for communication. For performance, battery consumption and reusability (of data) reasons, most application logic is executed on the server. Therefore, the application on the client side is lightweight and each application runs their own service on the server. The client communicates only over this service to the MobiSoC middleware. All location and social information is stored on the server. To protect the user's privacy, users are able to set rules on the server such as a certain user is only able to retrieve a user's location at a certain time span. Figure 2.3 shows the MobiSoC architecture. Two example applications are implemented. Tranzact is a place-centric MSCA. A useful scenario in Transact is a user wanting to know the menu in the cafeteria that is only announced in the cafeteria. The user's contacts that are currently located at the cafeteria are contacted to answer the question. Clarissa is a people-centric MSCA. An example scenario in Clarissa is, if a user wants to hang around with another person. Potential candidates must fulfill the following requirements: The candidate has to be physically close to the searcher and (s)he is a friend of the searcher or share common interests. The result is determined by sophisticated matching algorithms. [GKBB09]

Figure 2.3: MobiSoC architecture [GKBB09]: The MSCA services run on the server and include functionality to maintain the users' privacy, to notify the users, to collect data of people, places and locations and to learn the social state of people, places, people to people and people to places. All data are stored in a database.

2.4 MobiClique

MobiClique is "a middleware for mobile social networking" [POL+09]. It is able to create and maintain ad-hoc social networks for content and profile sharing via bluetooth or WLAN (IEEE 802.11) as seen in Figure 2.4. An ad hoc social network is limited to the physical proximity of the users. The user's profile including the friend list of existing social networks such as Facebook are used for MobiClique. Therefore, an internet connection is required at times to synchronize the user's profile. If MobiClique users are in physical proximity and share a relationship based on the profile and the friend list, the users will be alerted and can introduce each other, exchange content or create a friendship relation. Due to the lack of a server, the profiles are exchanged between the devices. Related devices form a network. Messages addressed to a specific user are routed through the network based on the friend list of each user. Thus, messages are forwarded in a single network hop or in multiple hops. The number of hops is therefore defined as the number of intermediate devices (routers) that are involved in the message exchange plus one. Group messages are flooded until every group member receives the message. Messages are discarded after the MobiClique TTL (time to live) has expired. The TTL consists of a time and a number of maximum hops. Applications are able to run on top of the middleware. The mobile social networking application displays all devices including their profiles in the neighborhood. Friendships can be established or removed. The asynchronous messaging application is used for messaging including sending texts or files. The epidemic newsgroups application is similar to usenet. [POL+09]

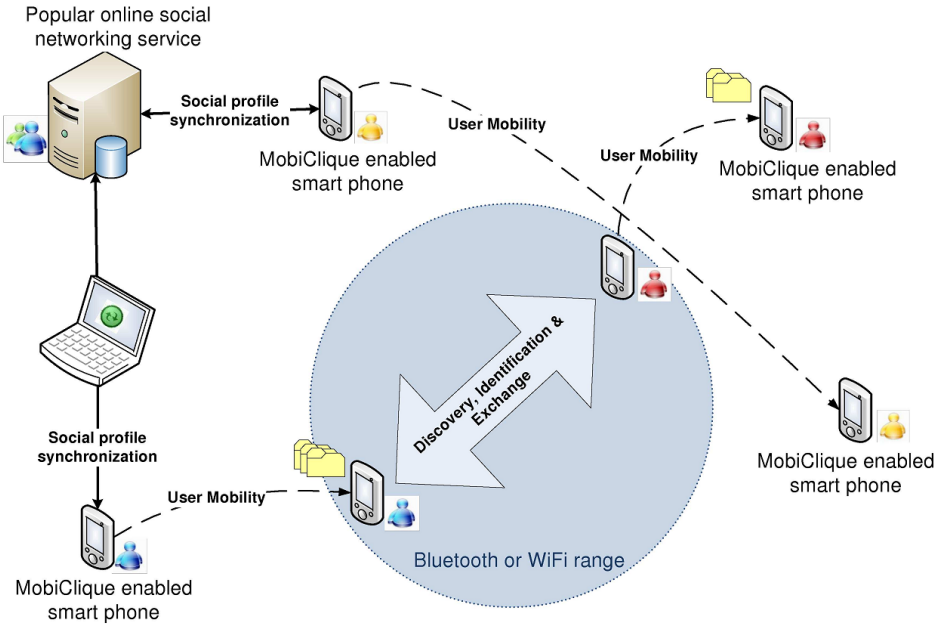


Figure 2.4: MobiClique system overview [POL+09]: The users retrieve their profile from existing social networks by connecting to the internet directly or via an internet enabled device. All other operations do not require an internet connection. Related users that are in the close proximity of each other are notified and are able to exchange their profiles, exchange content or create a friendship relation.

2.5 EZeeCom

EZeeCom [BN07] enables users to discover the whereabouts of the users and have spontaneous social interactions over mobile ad hoc networks using WLAN (IEEE 802.11b). It also enables a user to retrieve the approximate location of another user. All messages except location information are exchanged without a server. The location server is accessed over the LAN. It owns a location database to transform from symbolic locations (e.g., an address) to geometric locations (longitude and latitude) and vice versa. If a user is outdoor, the location will be retrieved by the GPS receiver. If a user is indoor, the location will be determined by sending the access point's MAC address to the location service. If a new user joins a network, it will send a hello message to all her/his neighbors. The neighbors acknowledge the hello message and forward the hello message to all neighbors. This is called the discovery phase. The hello message contains location and time information among others. Each user stores its neighbors and the location of all users. Messages are forwarded by flooding or by the location-based forwarding approach. Flooding is used to broadcast the hello messages. All other messages are forwarded based on the receiver's location. If an indoor user recognizes that (s)he has outdoor neighbors, the indoor user will be a router node and vice versa. The router nodes forward messages from indoor users to outdoor users and vice versa. Some applications were implemented that search for all users in the proximity and retrieve their brief profile and detailed information, search users with certain properties, send private messages, get an alert when a specific user is in the proximity and retrieve the location of a specific user. [BN07]

2.6 SAMOA

The Socially Aware and Mobile Architecture (SAMOA) [BMT07] enables users to create anytime, anywhere social networks. These networks group the users in the physical proximity that share the same interests. SAMOA provides egocentric social networks that search for nearby people of similar interests. There are three user roles. Managers (humans or software) are interested in creating new social networks by defining the discovery scope boundaries and the semantics. Clients are inside the discovery scope boundaries and potential members of the social network. Members are the users in a social network. Each user is allowed to play all user roles. SAMOA introduces the concept of a place. A place is the set of all clients that are close to the manager. It is restricted by the discovery scope boundaries and defined by the manager. Physical proximity is defined as a maximum number of network hops. Figure 2.5 visualizes the mapping from places onto a mobile ad-hoc network (MANET). Users are able to switch between places and can be in more than one place at any time. All places and users have their own profile. A place profile defines the activity of the place and its members. User profiles include personal information like age, gender and education, the preferred activities and the preferences of those activities. The underlying assumption is that places will influence user's activities. Furthermore, there is a discovery profile assigned by the manager to each place. It defines the required preferences for the users to become members of the social network. With the help of two consecutively executed semantic matching algorithms users are assigned to social networks. The first algorithm compares user profiles with the place profile. The second one compares user profiles with the discovery profile. SAMOA defines two types of social networks. The first one is a place-dependent social network that consists only of colocated users. The second one is a global social network that provides a history. The history is built of all user profiles in the manager's place-dependent social networks. A prototype that implements SAMOA was used in a book shop. SAMOA is used in this scenario to broadcast product advertisements. The book shop broadcasts a advertisement in its place-dependent social network to all present customers. All customers that visited the book shop in the past also receive the advertisement through the book shop's global social network. All customers are able to forward the advertisement in their respective place-dependent and global social

networks. UDP is used to deliver the messages in place-dependent social networks. Users advertise their presence via presence messages. If no presence messages of a certain user over a certain duration are received, the user will be removed from the place-dependent social network. [BMT07]

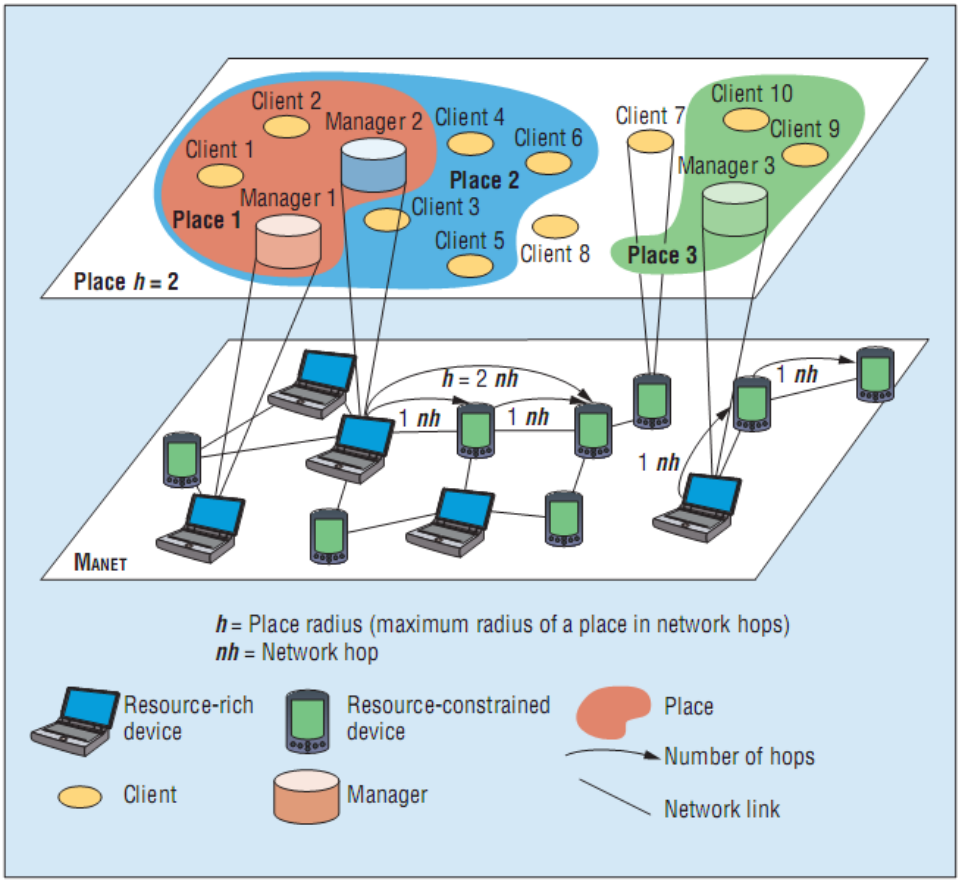


Figure 2.5: Example place mapping in SAMOA onto a mobile ad-hoc network (MANET) [BMT07]: Manager 1, manager 2 and manager 3 create three places with a place radius h of 2 network hops nh. Therefore, clients 1-2 are assigned to place 1, clients 1-6 are assigned to place 2 and client 9-10 are assigned to place 3. Clients 7-8 are not assigned to place 1 nor place 2, because the number of network hops between the correspondent managers and the clients is greater than 2.

2.7 RoadSpeak

RoadSpeak [SHSI08] enables users to establish voice chats between related users. The motivation for RoadSpeak is that people waste a lot of time driving between two places (e.g., the way to work). This time can be used for discussions with other drivers. In contrast to individual traffic, public transport enables people to communicate with each other. Therefore, Vehicular Social Networks (VSNs) are formed. VSNs can be created for entertainment, utility or emergency reasons. A VSN is created by an owner that defines a profile consisting of time, location (route) and interests. Those willing to join a VSN are assigned automatically to a VSN based on a matching between their profile and the VSN profile. However, there can be VSNs that do not depend on locations. Users are able to invite other users to VSNs. VSNs can be public or private (i.e., the group owner must approve new users), open or closed,

open or moderated and hidden or advertised. RoadSpeak is not only limited to mobile phones, because cars may be equipped with onboard computers that are connected to the internet. A global server is involved in the communication process, stores and manages all VSNs. Voice over IP and chat rooms are used for the message exchange. [SHSI08]

2.8 Conclusions

There are several approaches to realize spontaneous social interactions. Spontaneous P2P social networking, social computing applications, ad-hoc social networks, anytime and anywhere social networks, vehicular social networks, etc. have more or less the same idea, but with different aspects, for different purposes, under different conditions and circumstances and with different assumptions. The systems are compared and summed up with the help of the following criteria:

Network joining criteria: The presented systems can be divided into two categories: Location-centric (i.e., only people that are in the physical proximity to each other are considered to form a social network - SCOPE, MobiClique, EZeeCom, SAMOA) and global (i.e., in principle all people of the world are considered to form a social network - Mobilis, MobiSoC, RoadSpeak). This categorization has of course a huge impact on the purpose, design and architecture of the system. Location-centric social networks are used for augmenting the physical world with virtual features such as content sharing, to get to know of colocated persons or to use persons in the proximity of a user as a knowledge database. Global social networks are used to find new contacts, for content sharing, for collaboration or for entertainment. Obviously, all systems deal with the location of the users. Location is usually a property for getting assigned to a specific social network. Some systems also introduce matching algorithms based on properties such as interests, activity and time for assigning people to social networks.

Infrastructure required to realize the idea: Location-centric social networks operate on local area networks and usually do not use any servers. Messages can be exchanged by routing or broadcasting the messages among the users. Content can be stored for instance on super-nodes (SCOPE). However, a global server can be useful to protect the privacy of the users (EZeeCom). This category of social networks relies on ad-hoc WLAN (IEEE 802.11) or bluetooth. A problem with both communication standards is, that users usually turn them off for security and energy consumption reasons. Furthermore, many problems might occur during the establishment and the maintenance of these networks. Scalability and high energy consumption are still problematic, because each node has to forward (route) messages to other users and share its storage and computation capabilities with other users [CCL03]. The message routing approach raises important security and privacy problems and the storage and management of data and lists such as friend lists is difficult [SP06]. Global social networks use the internet as a communication channel and use servers for communication and management. Due to the low priced mobile internet solutions, the internet has found its way into nearly every PC or smartphone. Most users are always online. Nevertheless, connection losses and the high energy consumption is still a big problem on mobile devices. In foreign countries the expensive roaming fees are currently also a problem.

All presented systems are extensible and several applications are able to act on top of them. They do not implement one huge social network like social networking sites as Facebook and MySpace do, but they employ small social networks that are created on the fly. Currently a generic approach that can be used for all types of social interactions is missing. The current discovery methods for location-based social methods do not use the provided capabilities of the latest mobile devices. Currently there is no implementation of an SSN system on modern platforms like Android.

Chapter 3

Fundamentals

3.1 Social Networks

As already mentioned, social networks are a well understood field investigated in different research disciplines such as sociology, behavior recognition, anthropology, statistics, mathematics and computer science. There are several definitions for social networks:

> **DEFINITION A:** "A social network consists of a finite set or sets of actors and the relation or relations defined on them. The presence of relational information is a critical and defining feature of a social network." [WF94]

> **DEFINITION B:** "We define social network sites as web-based services that allow individuals to (1) construct a public or semi-public profile within a bounded system, (2) articulate a list of other users with whom they share a connection, and (3) view and traverse their list of connections and those made by others within the system. The nature and nomenclature of these connections may vary from site to site." [BE07]

> **DEFINITION C:** "Social networking websites are online communities of people who share interests and activities, and who are interested in exploring the interests and activities of others. They typically provide a variety of ways for users to interact, through chat, messaging, email, video, file-sharing, blogging, and discussion groups." [Han08]

> **DEFINITION D:** "A social network is a group of collaborating, and/or competing individuals or entities that are related to each other." [Zha10]

To understand DEFINITION A a few definitions are required. Wasserman and Faust [WF94] defines an actor as social entities - i.e., discrete individual, corporate or collective social units. A relation is the collection of ties of a certain kind among pairs of actors. Ties are the links among actors such as friendship, business transactions, participating in an event, knowledge, interest, migration, a road, authority or affinity. DEFINITION A leads to the underlying concept of social networks that is a social graph. It consists of nodes representing actors and edges representing social ties. In social sciences this graph is called a sociogram.

The small world problem [Mil67] leads to the six degrees of separation. That is, every person in the world is approximately six hops away from any other person in the world (i.e., friend of friend of ...) [Gua90]. Figure 3.1 shows two examples for social graphs. They have the same topology, but graph (b) emphasizes the analysis of the actor in the middle with the largest circle. The circles' sizes encode how many degrees the actors are away from the studied actor in the middle. For instance, the actor with the smallest circle is three degrees (i.e., three hops) away from the actor with the largest circle.

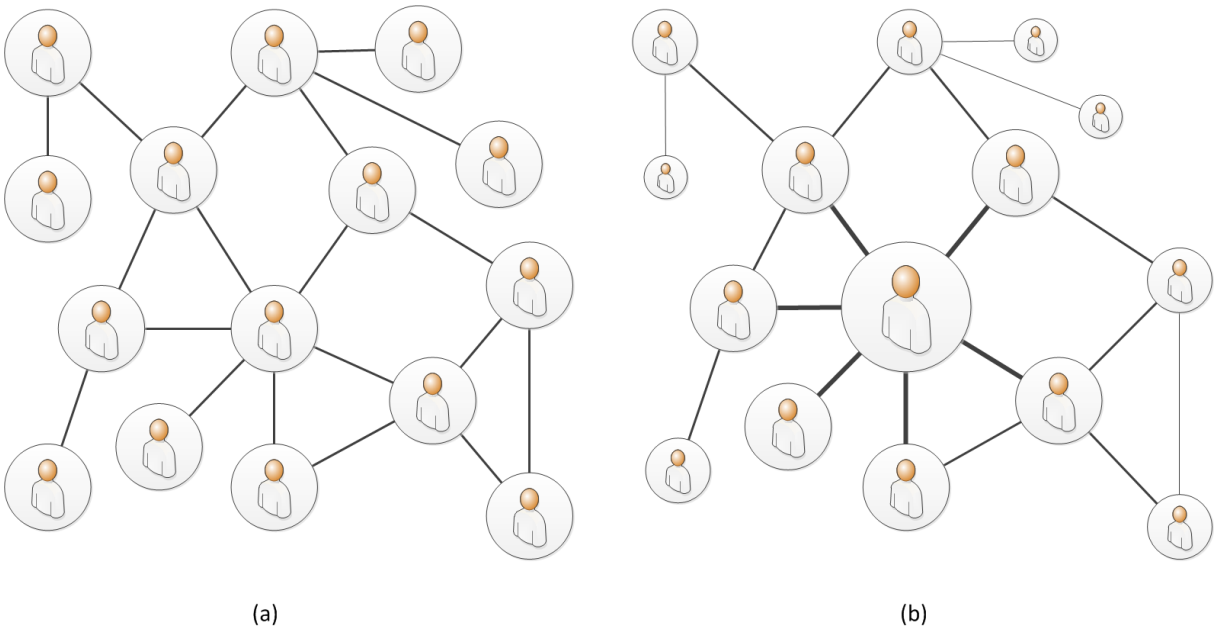


Figure 3.1: Social graph of a network [Gol08]. The graph of (a) and (b) are the same, but graph (b) emphasizes the analysis of the person in the middle with the largest circle. The actors are represented as nodes and the social ties as edges.

DEFINITION B and C are definitions for online social networks. They are typically accessed through websites. Online social networks are nowadays used for a wide variety of applications such as business, education, medicine and dating. They are used to publish and share content and media, find new contacts and maintain offline relationships. Many social networks have millions of users. In the last years online social networks have found their way into mobile devices. When mobile devices are involved in social networks, the term mobile social network is commonly used. DEFINITION D is a short version of the previous definitions.

Social Networks are much older than the PCs and the IT. In the late 19th century sociologists were precursors of the social network theory [Zha10]. Anthropologists and sociologists were the pioneers of social networks [WF94] in the second quarter of the 20th century. The term "social network" goes probably back to J. A. Barnes in 1954 [WF94]. At the beginning of the last quarter of the 20th century a research group at Harvard University published several articles and social networks became increasingly popular. Social networks were analyzed to study social and economic phenomena such as spreading of diseases and technologies. [Zha10]

In contrast to social networks, online social networks have a comparatively short history. The precursors of social networks were newsgroups, bulletin boards, dating and community sites. The first real social network was SixDegrees.com started in 1997 (discontinued in 2000). It offered personal profiles and friend lists that could be accessed by other friends. At the turn of the millennium AsianAvenue,

BlackPlanet and MiGente arose. They provided personal, professional and dating profiles. Friendship relations could be established without approval. In LiveJournal, launched in 1999, users could establish one-directional friendship connections to follow their journals. LunarStorm offered users 2001 friend lists, guestbooks and diary pages. At the beginning of the 21st century many new social networks were started. Friendster was one of the most popular social networks. It was launched in 2002 and wanted to compete with online dating sites. Users were not introduced to strangers, but users should get in contact with the friends of friends that were not more than four degrees away. Friendster annoyed its users by censorship (e.g., deleting fake profiles) and rumors occurred that Friendster would charge fees. Many Friendster users switched to MySpace. However, Friendster is still active. Many new social networks for dating, meeting people, business, etc. arose that last until now. Recently, many media sharing sites offer social networking features such as YouTube and Flickr. [BE07]

Communities and Local Government [Com08] state that the high popularity of online social networks is due to the networking factor (i.e., creation of networks with other related people) and creating and sharing content with others without creating and maintaining an own homepage. Social networks can be categorized based on Childnet [Chi08] into the following types (where a social network can be of more than one type):

- **Profile-based social networking services:** Are organized around profiles, e.g., Facebook, MySpace
- **Content-based social networking services:** Used for content sharing, e.g., Flickr, YouTube
- **White-label social networking services:** Mini-communities are formed for events, interests and activities (i.e., mini social networks in a huge social network)
- **Multi-User virtual environments:** Online virtual environments such as Second Life
- **Mobile social networking services:** Mobile phone versions of a social network, e.g., Facebook, Twitter
- **Micro-blogging / presence updates:** Publish status messages, e.g., Twitter
- **Social search:** Search across profiles of existing social networks

Nowadays, Facebook, MySpace, XING and Twitter are some of the most popular online social networks. Facebook [Fac11] was started 2004 for students of the Harvard university. As Facebook expanded, other university and school students were able to register at Facebook and in 2006 Facebook was opened for all people over the world. It offers a personal profile, bulletin boards (users are able to post text messages, status messages, etc.), news feed, video and photo galleries, groups, fan pages (one-directional relations), friend lists, event scheduling, a chat, private messaging and friendfinder. Developers are able to implement applications, that extend the functionality (e.g., social games, voting, quizzes, statistics) or link the social network with other sites, networks or services. MySpace [MyS11], launched in 2003, was formerly primarily designed for artists, musicians and their fans. It offers users custom styleable pages with a comment feature, friend lists, bulletin boards, an instant messenger, groups, a music section, applications similar to those in Facebook, a photo and video gallery. XING [XIN11], launched in 2003, is a social network for business and private contacts mainly in Europe. For some of XING's functionality a premium account that costs money is required. It offers personal profiles with the ability to post a curriculum vitae and upload credentials and certificates (therefore job searches and job offers are possible), event scheduling, discussion forums, groups, a job market, a messaging system and a search functionality. Twitter [Twi11], which started in 2006, is a microblogging platform. Users are able to maintain a

profile, post short text messages and retweet messages (i.e., post short messages of other users) on their profile page. Each user can follow microblogs of other users and send private messages.

3.2 Android

3.2.1 Overview

Over the last years mobile devices such as smartphones have become more and more popular. Whereas PCs offer a large amount of processing and storage power, mobile devices have limited processing and storage capabilities, limited input and output devices and very limited battery power. The requirements of mobile platforms / mobile operating systems are therefore different to operating systems on PCs. Android is an open and free mobile platform that is owned by the Open Handset Alliance. "The Open Handset Alliance is a group of 79 technology and mobile companies who have come together to accelerate innovation in mobile and offer consumers a richer, less expensive, and better mobile experience." [Ope11] One of the 79 group members is Google Inc., which is the main driving force behind Android. The Open Handset Alliance was founded on November 5, 2007. Android was announced on the same day. On September 23, 2008 the Android 1.0 SDK and the first hardware T-Mobile G1 was released [And11]. Android is released under the Apache Software License 2.0 with a few exceptions such as Linux kernel patches, which are released under the GNU General Public License 2.0. Therefore, there are no licensing limitations for application developers and Android can be adapted by hardware manufactures and developers.

At the end of 2010 Android was the world's leading smartphone platform [Alt11]. The most important competitors to Android are Symbian, iPhone OS, BlackBerry and Windows Mobile. Table 3.1 gives a brief overview of these platforms.

	Android	**Symbian**	**iPhone OS**	**BlackBerry**	**Windows Mobile**
Founded	2007	1998	2007	1995	2002
License	Open Source (Apache v2)	Open Source (EPL)	Proprietary	Proprietary	Proprietary
Programming Language	Java / C++	C++	Objective C	Java	.NET
Multitasking	Yes	Yes	Limited (not for 3rd party apps)	Yes	Yes
Underlying OS	Linux	Symbian	Darwin	Blackberry OS	Win32
Device Customers	Many	Nokia	Apple	RIM	Many

Table 3.1: Comparison of the most important mobile platforms [KP11]

The following sections provide the required Android fundamentals. Android's specific concepts are introduced that are required to understand this thesis. More complete information about Android is available in the Android Developer Guide [Goo11a], which is also a reference for this and the following sections.

3.2.2 Architecture

"Android is a software stack for mobile devices that includes an operating system, middleware and key applications." [Goo11a] The Android architecture is based on the Linux 2.6 kernel, which is used as a hardware abstraction layer for Android. The Linux kernel provides a driver model and other system services such as memory management, process management, network stack and a security model. The remainder of five layers altogether are: Android Runtime, Libraries, Application Framework and Applications. The discussed Android architecture is shown in Figure 3.2.

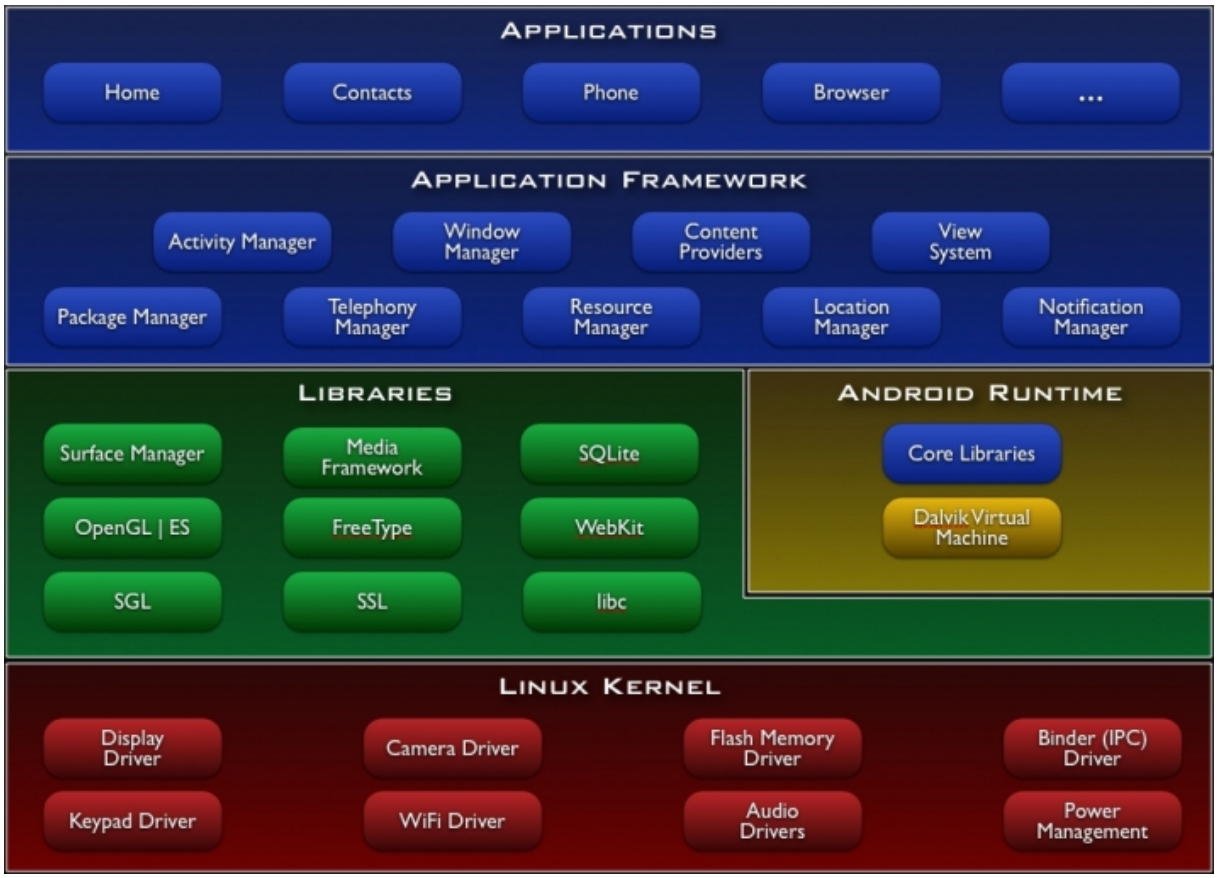


Figure 3.2: Android architecture [Goo11a]: Consists of the Linux kernel, libraries, the Android runtime, the application framework and the applications. The components on the top of the architecture take advantage of the components at lower levels.

On top of the Linux kernel, native libraries that are written in C / C++ are provided:

- **Surface Manager:** Manages the different windows for different applications
- **OpenGL ES:** Graphics library for 2D and 3D graphics; it is able to take advantage of hardware acceleration if available on the device
- **SGL:** Graphics library for 2D graphics; used by most of the applications
- **Media Framework:** Provides the video codecs, audio codecs and the image file formats such as MPEG4, MP3 and JPEG

- **FreeType:** For rendering bitmap and vector fonts
- **SSL:** For encrypting and decrypting data
- **SQLite:** Lightweight database engine; can be used by applications and is the core for most of the data storage
- **WebKit:** Open source browser engine
- **libc:** Standard C library; library for the programming language C; provides methods for I/O operations, math, strings, etc.

The Android runtime consists of core libraries and the Dalvik virtual machine (Dalvik VM). The core libraries are written in Java and bundle the functionality of the core libraries in Java such as collections, I/O operations, math and concurrency. The Dalvik virtual machine is designed for the requirements and limitations of mobile devices. In contrast to the Java virtual machine, Dalvik is register-based. It runs dex-files (abbr. for Dalvik executable). Dex-files are byte codes that are highly optimized in matters of memory and runtime. Multiple Java class files are converted into a single dex file by the dx tool. There is one Dalvik virtual machine for each process. Usually each application runs in its own process. The Dalvik VM includes a garbage collector like the Java VM.

The application framework takes advantage of the functionality provided by the libraries. It is used by all applications and helps developers to build applications easily. The application framework is written in Java and consists of:

- **Activity Manager:** Manages the lifecycle and navigation between applications and their components
- **Package Manager:** Keeps track of the installed applications and their capabilities
- **Window Manager:** Manages the windows
- **Telephony Manager:** API for the phone functionality
- **Content Providers:** Allows to share data across applications
- **Resource Manager:** Stores localized strings, drawables, arrays, layouts, etc.
- **View System:** Contains the UI elements and layouts; handles drawing and event dispatching
- **Location Manager:** Manager for geographic locations; gets the user's location or registers for intents that are sent when the user is close to another user, buildings, shops, etc. based on proximity
- **Notification Manager:** Manages notification to the user that are displayed in the status bar

The actual applications visible to the end user are implemented on top of the architecture. Android typically contains a set of standard applications such as the browser, email client, home screen, contacts and phone. However, device customers and developers building customized Android versions are able to replace or remove applications. Furthermore, users are able to install third party applications. They are also able to replace the standard Android applications without uninstalling them. Each application is able to use public features and data of other applications.

3.2.3 Applications and Development

Each application is bundled into an apk-file. The apk-file contains the dex-files, all resources (e.g., files, images, videos, strings, layouts) and a manifest file. The Android manifest file is an XML file that defines all application components, used permissions, created permissions and metadata such as the resource ID of the application icon, application version, title, package, supported screens and required hardware of the application. Each apk-file is signed with a certificate to identify the application developer. The application developer owns the certificate's private key. Applications can be downloaded from markets such as the Android Market, uploaded from a PC or downloaded from the internet.

Android applications are usually written in Java using the Android Software Development Kit (SDK). The Android SDK contains the Java APIs and useful developing tools (e.g., an emulator, debugging tools and testing tools). Some parts of an application might run faster on native code and therefore can be implemented using the Android Native Development Kit (NDK). These application parts are implemented in native programming languages such as C and C++. The native application methods are called through the Java Native Interface (JNI).

3.2.4 Application Components

An Android application consists of activities, services, broadcast receivers and content providers. Not all applications must contain all of the mentioned components. For instance, many applications consist only of activities.

An **activity** is a visual screen that is displayed to the user. Most applications consist of multiple activities, each implementing a user interface screen. Examples for activities are a user interface for composing a new e-mail, a home screen of an application that presents the most important functions, a contact list for an instant messenger, a highscore of a game or a game itself.

A **service** is a long-term task that runs in the background. A service does not display anything on the screen except of notifications on the notification bar. There are two types of services: started and bound. However, a service can also be started and bound. A "started" service is started by an application component, executes the intended work without communication with the application component which started the service and is usually stopped by itself when the intended work has finished. At a bounded service an application component is able to bind to a running service or the service is started before binding to it. After the binding the application component can communicate with the service. The communication works via an interface that declares methods for the provided functionalities of the service. There are two kinds of services, which are local and remote services. A local service runs in the same process as the application. Only activities of this application are able to bind to a local service. In contrast, a remote service runs in its own process and if the developer of this service marks it as "exported" all applications can bind to this remote service. Examples that are typically done in a service are playing music, holding a connection to a server for certain purposes such as instant messenging or exchanging data.

A **broadcast receiver** listens for specific system or application events. If the broadcast receiver receives such an event, it will execute a predefined functionality such as notifying the user, launching an activity, binding to a service or process and store data. Examples for system broadcast events are the completion of the phone boot process, the installation, uninstallation or update of an application, the change of the battery state, or the change of the current time zone.

With the help of a **content provider** some application data can be shared to other applications. Android declares predefined content providers for video, images, audio and personal contact information.

A couple of the mentioned Android components are started and linked together with a concept called **intent**. An intent is an asynchronous message that contains the description of the operation to be performed. Furthermore, it can contain additional data. An activity is launched with the help of an intent. To start or bind to a service an intent is required. A broadcast is also realized via an intent by sending an intent from an activity or a service to all interested broadcast receivers. Usually intents are used for asynchronous communication between application components. In addition activities are able to communicate synchronously. That is, after executing the called activity a result consisting of an intent, a result code and a request code is pushed to the calling activity and the calling activity is resumed. Intents are categorized into explicit and implicit intents. An explicit intent specifies the exact application component (class) to be run. In contrast, for an implicit intent Android finds the best application, respectively the best application component, based on the action, type, URI and categories specified in the intent. If Android finds more than one application component that matches the intent and cannot determine the best application, it will ask the user to select the best application. Thus, applications can be replaced. Application developers are able to use a hybrid approach in between explicit and implicit intents by asking the system for all application components that are able to handle a certain intent. Afterwards the developer chooses a suitable application component and sends an explicit intent. As mentioned, all application components are specified in the Android manifest file. Each component can be declared as public (i.e., all applications are able to use the component) or private, and intent filters may be declared. The intent filter of a component tells Android which implicit intents should be passed to the component. Examples for intents are show the specified video, pick a photo, show a website, show a location, show the detail page on the Android Market for a specific application or open the buddy list in an instant messenger application.

When the activity's work has finished, the user might be redirected to the last executed activity automatically. Furthermore, the user is able to navigate back to previously executed activities by pressing the back key. A stack is used for bootstrapping the executed activities. The current visible activity is on top of the stack. During navigation the activities are pushed (i.e., a new activity is visible) onto the stack and popped (i.e., the user navigates back or an activity has finished) from the stack. The activities used for an application are grouped into a task. Each task is built using its own stack. Not all activities of a task must belong to the application, because each application is able to execute public components of other applications. The user is able to navigate between all running tasks with the help of a task manager. Android automatically removes tasks that are running for a long time in the background. That is, only the activity on the bottom (i.e., the root/start activity) is recreated/resumed when the user returns to the task.

3.2.5 Permissions

A key concept in Android's security model is the use of permissions. For security relevant operations the application developer must acquire a permission, which is declared in the application's manifest file. Examples for potentially dangerous operations are those that cost money (send an SMS, initiate a phone call etc.), accessing the internet, camera, location of the user, use bluetooth or read the user's contacts. The user must accept all permissions, before the installation of the application. The user is not asked for granting a permission at runtime. Each application is able to acquire existing permissions and create new permissions. Each application component can declare a permission in the manifest file to restrict access to only those applications that have acquired the declared permission in their Android manifest file. There are different types of permissions. Permissions of type normal are permissions that have only a minimal risk for the user or the system such as using the vibrator, retrieving the network state and receiving a broadcast event after the device has booted. The user need not accept this type

of permissions explicitly, even though the user is able to see permissions of type normal on request. Permissions of type dangerous are much more riskier than permissions of type normal and have to be accepted by the user before the installation of the application. Signature-level permissions are typically used to protect application components from being used from applications of other developers. An application will only be able to own a signature-level permission (i.e., declare it in the Android manifest file) if the caller's application package is signed with the same certificate than the callee's application. Otherwise a `SecurityException` is thrown. Permissions of type "signature or system" are similar to signature-level permissions except that the system is also granted to use the secured component. [Bur09]

3.2.6 Lifecycle

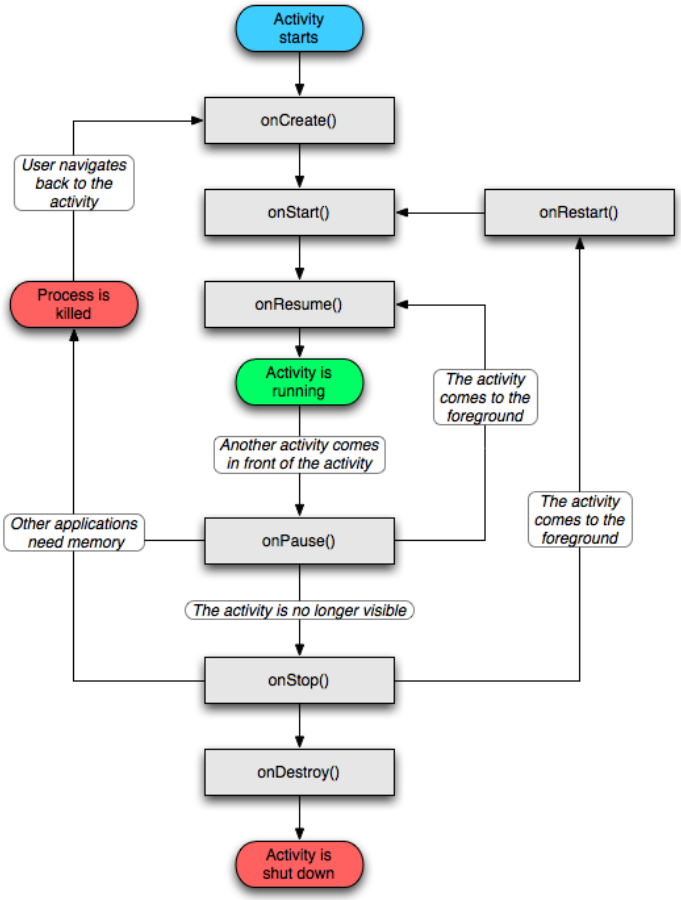
Each application component in Android has a lifecycle from the instantiation until the dead of the component. The lifecycle on mobile devices is very important and comes into play especially when asynchronous events (like a phone call) occur, the memory runs low or the user navigates between activities. Each component has gotten a different lifecycle.

The **activity lifecycle** has gotten the states running, paused and stopped. An activity is running as long as it is visible to the user. A paused activity is still visible, but the user focuses on another activity. This happens when the focused activity does not fill the whole screen or the activity is transparent. A paused activity can be resumed (i.e., it gets focus), stopped or killed if the system memory runs low dramatically. A stopped activity is not longer visible. It can be restarted (e.g., when the activity was not killed and the user navigates back to the stopped activity), killed when the system needs memory or destroyed when the user or the activity itself signalizes that the activity is no longer needed. When the activity is killed, the activity state might be saved and when the activity is recreated the activity state might be restored. Figure 3.3 (a) shows the activity's lifecycle.

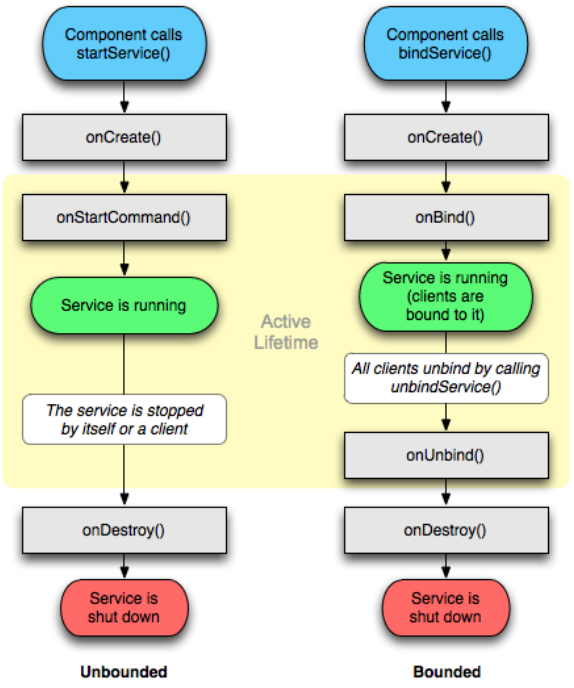
The **service lifecycle** is much simpler than the activity lifecycle. The lifecycle of the "started" (unbounded) and "bounded" service is different. A "started" service is started by some component, executes the intended work and is stopped by a component or by itself. A "bounded" service is started automatically when the first application component binds to it. Then the application components are able to communicate with the service. It is stopped when the last bound application component unbinds. Figure 3.3 (b) shows the service lifecycle of the "started" (unbounded) and "bounded" service.

The **broadcast receiver lifecycle** is the simplest one. A broadcast receiver is only running if it receives a broadcast message and processes it. However, the process of an application that contains a broadcast receiver is always running and is protected from being killed.

The application developers have to take special care of the Android lifecycle. The lifecycle will only work correctly, if the developers implement the corresponding lifecycle methods based on the capabilities and requirements of the application. For each mentioned lifecycle state, a method that should be overridden by the application developer exists.



(a) Activity Lifecycle



(b) Service Lifecycle

Figure 3.3: Activity lifecycle (a) and lifecycle of the "started" (unbounded) and "bound" service (b) [Goo11a]

3.2.7 Interprocess Communication

Remote Procedure Call (RPC) usually means to call a method that is executed remotely on a different machine. There are several frameworks such as Java RMI or CORBA to realize remote procedure calls. In contrast, RPC on Android means to call a method that is executed remotely in a different process on the same device. Therefore, it is called inter process communication (IPC) on Android. IPC is very important concerning remote services. Each remote service and each application runs in its own process. An application has to use IPC to call the remote service's methods.

All the parameters of such a remote method have to be marshalled by serialization and unmarshalled by deserialization. That is, all data is copied from the local process and address space to the remote process and the remote address space. If there is a return value, it will be transmitted in the opposite direction back to the caller's process. The stubs are the local proxies of the remote methods and are responsible for marshalling and unmarshalling. They are automatically generated by the aidl tool that is provided by the Android SDK.

Android uses OpenBinder [Pal11] for IPC instead of the standard Linux IPC. OpenBinder uses a custom kernel module. Each of the two processes involved in the IPC owns a thread pool that is responsible for executing, sending and receiving inter process calls. These thread pools are managed by the custom kernel module. [Pal11]

Interfaces are used to describe the capabilities of the remote services. These interfaces are defined by an interface definition language (IDL) called Android interface definition language (AIDL) similar to CORBA. Of course, the remote service and the client must own the AIDL interface. The AIDL supports primitive Java data types, strings, lists (including generic lists consisting of strings, AIDL interfaces and `Parcelables`), non-generic maps, `CharSequence`, AIDL interfaces and custom parcelable class. With the help of the parcelable class Java objects can be sent from one process to another. In contrast to other AIDL interfaces that are passed by reference, `Parcelables` are passed by value. Therefore, changes to the object in the local process are not transfered to the object in the remote process. `Parcelables` must implement the `Parcelable` interface and methods for the serialization process. Android does not use Java serialization for IPC due to performance issues [HKM10]. Instead, `Parcelable` is used.

Android IPC is very lightweight and fast, but has some limitations:

1. Exceptions are not supported.
2. Method overloading is not supported.
3. Generics are not supported (with a few exceptions).
4. All remote method calls are synchronous. That is, the caller's thread is blocked until the remote method call has finished.

A solution for the first limitation would be to return a dummy value (e.g., a boolean) that indicates if an error occurred. This makes of course problems if, e.g., null values are allowed. A second solution to this problem is to use callbacks. A callback object is passed as an additional parameter. The method does not return a value. The callback object implements one method for signalizing that all went well and a second method for signalizing that an error occurred in the method. If the method should return a value, the return value is passed as a parameter to the "all went well" method.

A solution for limitation four is to spawn a worker thread in the remote service that handles the execution of IPC calls. This is often desirable, since the main thread of an activity must not be blocked. A blocking activity cannot respond to user interactions and system calls and is therefore shut down by the system.

3.3 XMPP

3.3.1 Overview

The Extensible Messaging and Presence Protocol (XMPP) is basically an instant messaging protocol. Instant messaging in its simplest form means to exchange messages between two persons (clients) in close-to-realtime, peer-to-peer manner. For scalability, security and management reasons, one or more servers are involved in the message exchange. XMPP sends messages in the Extensible Markup Language (XML) format. Each of the clients gets informed, when other known clients are online and ready to exchange messages. Therefore, presence messages are exchanged between the clients to signalize the online status of each client. To manage the presence information and display the contacts, a contact list is used. The contact list is called roster in XMPP. The formal description of RFC 3920 states: XMPP is "a protocol for streaming Extensible Markup Language (XML) elements in order to exchange structured information in close to real time between any two network endpoints." [SA04a]. XMPP defines several wire protocols. The XMPP core protocols are specified in RFC 3920 [SA04a] and RFC 3921 [SA04b]. They define the architecture, message format, message exchange, addressing, security, errors, presence and the roster. To extend the XMPP's core protocols new extensions, called XMPP Extension Protocols (XEPs), can be submitted to a consortium that reviews them and might consider publication of the new extensions. XMPP has a lot of great features that demonstrate the strengths of this protocol [SAST09]:

- **Standardization and Spreading:** XMPP is an approved standard by the Internet Engineering Task Force (IETF) since October 2004 and is used over more than 10 years. There are many servers, code libraries and clients for nearly every platform. XMPP has a large community.
- **Openness:** All standards are open and well documented. Everybody can use XMPP for free for any purpose and everybody is able to contribute to XMPP.
- **Security:** XMPP uses channel encryption and strong authentication to secure the message exchange.
- **Decentralization:** There is not a single server. Therefore, there is no single point of failure. XMPP is organized in small subnets with one server per subnet. Everybody is able to run an XMPP server.
- **Extensibility:** There are a lot of extensions that extend the functions of the XMPP core protocols. Everybody is able to develop new extensions.

In the mid 1990s instant messaging protocols were getting more and more popular. Many new instant messaging protocols were released. Unfortunately, all of the instant messaging protocols relied on proprietary protocols. These protocols were typically closed-source and undocumented. Users of one network were not able to talk to users of another network and developers had to rely on the given networks and infrastructure of the inventors. Therefore, it was not very appealing to develop applications for or on top of these protocols. Jeremie Miller took up these disadvantages and invented Jabber in 1998. In January 1999 Jabber was announced. A community was formed to push the development of Jabber. The community implemented a server, clients, protocols and extensions. In May 2000 the first version of the

first XMPP server (jabberd) was released and the core protocols were stabilized. The Jabber Software Foundation (JSF) was founded in August 2001 to coordinate the different developers and organizations, manage the core protocols and develop new extensions. In the next year the JSF submitted the core protocols to the Internet Engineering Task Force (IETF). The IETF decided to form a working group with the name Extensible Messaging and Presence Protocol (XMPP). Later, this working group was approved by the Internet Engineering Steering Group (IESG) and the core protocols were contributed to the Internet Standard Process. The working group refined and improved the protocols by security and internationalization in 2003. In October 2004 the core protocol and the instant messaging specifications were approved as proposed standards and were published as RFCs. The extensions were renamed from Jabber Enhancement Proposals (JEP) to XMPP Extension Protocols (XEPs) in the last quarter of 2006. In 2007 the Jabber Software Foundation changed its name to XMPP Standards Foundation (XSF). Over the years a lot of extensions, server implementations, instant messenger applications and many client libraries have been developed. [SAST09, Mof10, XSF11a]

XMPP has a wide range of application fields and large companies using and supporting it. Google Talk - Googles instant messaging system including voice and video chat - which was launched in August 2005, is based on XMPP. Facebook's chat feature is also based on XMPP. XMPP is used for multiplayer games such as the chess game Chesspark [SAST09]. Furthermore, it can be used for collaboration applications, social applications, monitoring, file and data exchange, notifications, etc.

3.3.2 Architecture

XMPP consists of a decentralized client-server architecture. Thus, there is more than a single server and therefore not a potential single point of failure. XMPP consists of many subnets. Each of these subnets has a single XMPP server that is addressed by a domain identifier. Usually each XMPP server is owned by a different organization or person. A client has to register at a single XMPP server and is a part of the corresponding subnet. There is no communication between clients. Instead, all messages between clients are sent via the XMPP server that is responsible for the sender. Due to explanation reasons, we call this server the sender's home server. The XMPP server will forward the messages to the recipient if the recipient is in the same subnet as the sender. Otherwise, it will look up the home server of the recipient through methods such as the Domain Name System (DNS), will forward the message to the recipient's home server and the recipient's home server will deliver the message to the recipient. This is similar to email, except of the fact that in XMPP the receiver's home server is contacted directly by the sender's home server. In the email system the sender's home server may forward the message to other mail servers than the recipient's home server in between. The communication protocol for the client-to-server (c2s) connections and for the server-to-server (s2s) is TCP. An XMPP server may deny s2s connections. If this is the case, this XMPP subnet will be isolated from the whole XMPP network on the internet or this network will operate only in a local area network (LAN) such as a company's intranet. Beside server and clients, there is an additional network component called gateway. The gateway operates as a translator to other messaging protocols such as email, IRC, SMS, ICQ and MSN. Figure 3.4 shows the architecture of XMPP.

3.3.3 Presence and Roster

Presence messages are used to broadcast the online status of a user. If user A wants to be notified of the online status of user B, user A will subscribe to user's B presence. User B has to approve the subscription request. Usually user B is also interested in user's A presence. So he sends a presence subscription

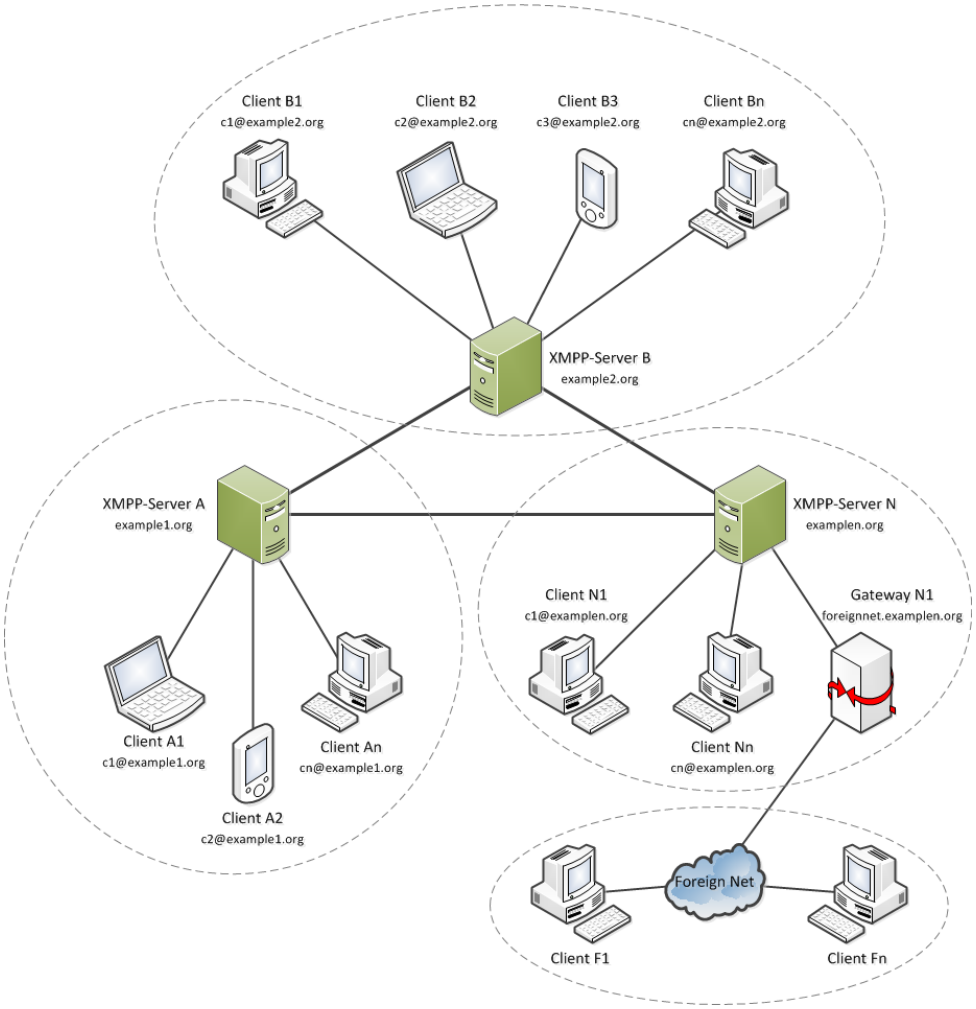


Figure 3.4: XMPP architecture [SA04a, SAST09]: Consists of servers, clients and gateways. A server is responsible for several clients and manages the message exchange between its clients and the clients of other servers. The gateway is a connector to other messaging protocols.

request to user A. If user A approves the presence subscription request, both users will exchange their online status. If a user goes online or offline, the user will send a presence message to the server and the server broadcasts the online status to all subscribed users. The server is also able to recognize when a client-to-server connection is terminated in an unusual way and therefore broadcast a presence message.

The roster is the contact list in XMPP. It stores the user's buddies and the online status of each buddy. The roster is stored on the XMPP server. The contacts in the roster can be grouped into roster groups to categorize and keep track of a huge number of buddies. Furthermore, roster groups are useful for securing PubSub nodes, sending group messages, blocking and filtering of messages, etc.

3.3.4 JID and Resources

A Jabber Identifier (JID) identifies any entity (e.g., server, user, bot, service) within the XMPP network. It is similar to an email address. A full JID consists of three parts:

1. **Node Identifier:** A name that identifies the endpoint. In case of a user, the node identifier is a nickname chosen by the user.
2. **Domain Identifier:** The XMPP server or a service where the endpoint is registered such as xmpp.org.
3. **Resource Identifier:** Identifies a specific connection (e.g., location, device, application) such as phone, fridge, home.

In contrast, a bare JID consists only of the node identifier and the domain identifier. An example for a full JID is alice@xmpp.org/phone, whereas the corresponding bare JID is alice@xmpp.org. With the help of the resource a user is able to have multiple connections to the server with the same JID (same login data). Each resource can specify its priority. The messages that are addressed to the bare JID of an endpoint are delivered to the resource with the highest priority.

3.3.5 Bots, Components and Services

A (client) bot is a client that is fully controlled by a program and not by a user. Depending on the intended use it may respond to messages (e.g., for gaming, data acquisition), display the message data on output devices, create content on the web, etc. Bots are usually easy and fast to implement. The problem with bots is that they are ordinary users and therefore do not scale very well due to the roster. For large systems the roster easily becomes too large and costly to manage and to retrieve [SAST09].

To solve this problem, server plugins can be written. Server plugins are coupled tightly to the used XMPP server and therefore have to be written in the programming language of the server. In contrast, external components can be written in any programming language and may run on a different server. The communication protocol between the server and the component is the Jabber Component Protocol [XSF05]. Every XMPP server usually provides different components (services) for supported extensions such as Publish-Subscribe and Multi User Chat (MUC). All mentioned components are addressed by a subdomain of the server's domain such as muc.xmpp.org. There is no direct communication between a component and a client. Instead, the server routes the messages based on the address.

3.3.6 Messages

Messages are exchanged asynchronously using XMPP. Thus, clients can send multiple messages to the server instead of waiting for the reply of the first message. Instead of polling the server for new messages, the server is able to push new messages to the client. This approach reduces the message count rigorously and therefore increases performance and scalability.

The message exchange between the client and the server works by default over a long-lived TCP connection. Messages in the XML format are exchanged (streamed) on the TCP connection. There is one XML stream from the sender to the receiver and one XML stream from the receiver to the sender. The authentication process is secured by the Simple Authentication and Security Layer (SASL) and the connection is typically secured by Transport Layer Security (TLS). The XML tag <stream> is the root (start) tag of an XML stream. A message is also called a (XML) stanza in XMPP. It is the direct child element of the <stream> element and may have child elements itself. The stanzas of the XMPP core protocol are:

- **<message>:** "can be seen as a 'push' mechanism whereby one entity pushes information to another entity" [SA04a]. Listing 3.1 shows the message stanzas during a single message exchange. The

first shown stanza is the message that maxmuster sends. The second one shows the stanza that the recipient receives. The second stanza is sent from the server on behalf of the sender.

- **<presence>:** It contains the presence (availability) of an entity and is broadcasted by the server based on the entity's roster. Furthermore, presence subscription requests, unsubscription requests and acknowledgments are sent as presence stanzas. Listing 3.2 shows the presence stanza after the login of maxmuster. The server sends the presence stanza on behalf of the sender.
- **<iq>:** The Info/Query (IQ) stanza is the only stanza where the sender always receives a reply. It is used to retrieve (get) or set data. The reply (result) message contains the requested data, a success message or is an error message. The sender of an IQ stanza must provide a unique ID. The reply contains the same ID as the sent stanza to associate the reply with the sent stanza. Listing 3.3 shows the IQ stanza to retrieve the roster. The second stanza is the reply (result) from the server.

```
<message to="marymuster@xmpp.org/phone" type="chat">
   <body>How are you?</body>
</message>

<message from="maxmuster@xmpp.org/home" to="marymuster@xmpp.org/phone" type="chat">
   <body>How are you?</body>
</message>
```

Listing 3.1: Maxmuster sends a chat message to marymuster. The second message is the message marymuster receives.

```
<presence from="maxmuster@xmpp.org/home" to="marymuster@xmpp.org" />
```

Listing 3.2: Presence stanza after the login of maxmuster sent from the server to marymuster

```
<iq type="get" id="mir_18">
   <query xmlns="jabber:iq:roster" />
</iq>

<iq to="test@blindeshendl.cg.tuwien.ac.at/home" id="mir_18" type="result">
   <query xmlns="jabber:iq:roster">
      <item subscription="both" jid="marymuster@xmpp.org">
      </item>
      <item subscription="both" name="max" jid="maxmuster@xmpp.org">
         <group>work</group>
      </item>
   </query>
</iq>
```

Listing 3.3: IQ stanza to retrieve the roster and the result from the server

Besides stream errors (e.g., connection lost, timeout), stanza errors might occur. Due to the fact that message stanzas are not acknowledged, stanza errors are the only way to notify the sender that something went wrong. Examples for stanza errors are that the receiver does not exist, the sender has not the permission to execute the action, the sent XML is malformed or an internal server error occurred.

3.4 XMPP Extensions

One of the strengths of XMPP is its extensibility. As mentioned, the functionality of the XMPP core protocols are specified as XMPP Extension Protocols (XEPs). There are only a few extensions that

are final, many extensions reached the draft status and many more extensions have other states such as experimental, historical, deferred and deprecated. Everybody can develop new extensions that are published publicly or are private to the developers. The types and states of the XEPs and the XSF standardization process for XEPs is documented in XEP-0001 [XSF10d]. There are extensions for nearly every purpose such as a multi user chat, registration of new users over XMPP, date, time and location formats, file transfer, voice and video calls. Most extensions need to be supported by the XMPP server. In the following sections the most important XEPs for this thesis are explained.

3.4.1 Service Discovery

The Service Discovery extension is used to discover the XMPP network and is specified in XEP-0030 [XSF08b]. It defines two IQ messages. The disco#info message is used for retrieving the features, capabilities, supported protocols and the identity of an entity. The disco#items message is used for requesting the items (e.g., services, nodes) of an entity.

3.4.2 Privacy Lists

The Privacy Lists extension is used to block messages in XMPP and enhances the defined privacy lists protocol of the XMPP core specification (RFC 3921) [SA04b]. It is specified in XEP-0016 [XSF07]. Any XMPP entity is able to define privacy lists. The server filters all messages based on the rules defined in the privacy list. All communication, only chat messages, incoming and/or outgoing presence messages or IQ messages can be blocked based on the JID, roster group, presence subscription type or globally. Each rule defines if it denies or allows communication, which messages should be blocked/allowed and the order in the privacy list. Each message is matched against the privacy rules according to the order in the privacy list. If a message matches a rule that denies communication, the message will not be delivered to the receiver. There are a default privacy list and privacy lists which are defined per resource or session. However, there is only one active list. That is, there is no layering of privacy lists. Furthermore, privacy lists can be deleted or modified at anytime.

3.4.3 Publish-Subscribe

The Publish-Subscribe (PubSub) extension is the representation of the PubSub paradigm in XMPP and is specified in XEP-0060 [XSF10b]. PubSub is used to distribute messages to an audience that is not necessarily known by the sender. To achieve this, a message distributor - called PubSub node - is used. All messages that are sent from the publisher to a PubSub node are broadcasted to all subscribers. Each PubSub node refers to a specific type or class of messages. The PubSub nodes are located at the PubSub service on the XMPP server. PubSub has got a large application field such as notifications of published posts on a website, location sharing, news feeds and sharing any kind of data.

Any addressable *entity* (e.g., user, bot, service, component) is able to use the features of the PubSub extension. If an *item* is published to a node, an *event* will occur and a *notification* will be sent to all subscribers. The notification typically contains a *payload*. The payload is the data of the published item in the XML format. Each payload has a *payload type* that is specified by the XML namespace. Each node is addressed by a bare JID and a node id or a full JID. In the latter case the resource identifier of the full JID is the node id and the domain identifier is the JID of the PubSub service. The PubSub service is usually addressed in the format pubsub.<domain identifier of the XMPP server>. The node id is unique at the PubSub service.

The PubSub extension provides a lot more features than simply broadcasting messages from a publisher to an interested audience. For instance, affiliations are used to manage permissions. The affiliations are organized in a hierarchy where each affiliation inherits the permission of the ancestors. The most important affiliation is the (node) owner, which is the root node of the affiliation hierarchy. The owner is the administrator of the node. The node creator is usually the owner, but not necessarily, because the node owner is able to manage affiliations. Therefore, (s)he is able to assign new owners and remove current owners. According to the PubSub specification, we will use the singular word owner, even though there can be more than one owner! Each PubSub node can be configured by the owner to match the requirements of the users or the used application. The following list gives an overview of the most important features introduced in the PubSub extension:

- **Message Storage:** A node can be configured to store the last, the latest or a specified number of messages. A subscriber is able to retrieve the stored items anytime. Publishers are allowed to delete stored items.
- **Security:** To secure the node data from unauthorized access, subscribers and publishers can be filtered with the help of the following models. Subscriptions to a node may be disallowed.
  - **Access Model:** The node access model defines who is allowed to subscribe to the node.
    - **Open:** Anybody is allowed to subscribe.
    - **Authorize:** Subscribers must be approved by the node owner.
    - **Whitelist:** Anybody that is on the whitelist managed by the owner is allowed to subscribe.
    - **Roster:** Anybody that is in a specified roster group of the owner is allowed to subscribe.
    - **Presence:** Anybody that is able to receive the owner's presence messages is allowed to subscribe.
  - **Publish Model:** The publish model defines who is allowed to publish to a node. Therefore, a publisher at a specific PubSub node need not own the publisher privilege if the publish model is "open" or "subscribers".
    - **Open:** Anybody is allowed to publish.
    - **Subscribers:** Only users that are allowed to subscribe to the node are allowed to publish.
    - **Publishers:** Only users that own the publisher privilege (these are the users with affiliation "owner", "publisher" and "publish-only") are allowed to publish.
  - **Blacklist:** A user that owns the outcast affiliation does not have any privileges on the node. The owner is able to assign or remove a user from the outcast affiliation.
- **Node management:** The owner is able to delete and modify the configuration of the node anytime. Furthermore, he is able to delete some or all items of the node, manage subscriptions and affiliations.
- **Privacy:** The publisher of an event is usually not included in the item. The PubSub extension specifies an optional feature called "Associating Events and Payloads with the Generating Entity". Depending on the node configuration, the publisher is included in the message for owners or publishers and owners.
- **Collections:** Collection nodes are hierarchically linked to other collection or leaf nodes. Leaf nodes are the standard PubSub nodes, that contain published items. In contrast, collection nodes do not contain any published items. If a user subscribes to a collection node, the user will receive

all notifications of the leaf nodes which are associated with the collection node. A subscriber may specify a subscription depth in her/his subscription request. The subscriber is not notified of events that occur in the leaf nodes having a depth greater than the specified subscription depth. Cycles in the resultant node graph are not allowed. Collection nodes are specified in XEP-0248 [XSF10e].

- **Notifications:** Depending on the node configuration, the subscribers receive notifications when items are published, items are deleted and/or the node is deleted. It also depends on the node configuration if the payload is included in the notifications when new items are published.

3.4.4 Personal Eventing Protocol

The Personal Eventing Protocol (PEP) is a user related, simplified and more user-friendly subset of the PubSub extension and is specified in XEP-0163 [XSF10c]. Each user has her/his own virtual PubSub service to create PEP nodes and publish events to other users. The users do not have to subscribe to a node explicitly. Instead of that, all users that are subscribed to a specific user's presence, are subscribed to all or a certain number of PEP nodes of the specific user. The latter is based on a filtering mechanism defined in Entity Capabilities (XEP-0115) [XSF08a]. With the help of the Entity Capabilities extension an entity announces its capabilities, because not every entity supports each extension. The capabilities are announced in the presence messages as a verification string. The verification string is a hash value of all capabilities among others. An entity that has not cached the verification string and therefore does not know the underlying capabilities, will send a disco info request to the sender of the presence message to get the capabilities. Concerning PEP, the interest in a specific event (payload) type is treated as a capability. Examples for the use of PEP are to share the user's current location, watched video, listened song and private information such as the user's bank account number, public key and contact information.

Chapter 4

Design and Concept

4.1 Outline

As already mentioned in Section 1.1, SSNs are small, well-defined and short-lived networks that enable spontaneous social interactions. An SSN application works on top of an SSN and enables the use of SSNs. A user can be a member of more than one SSN at the same time, even if several of these SSNs belong to the same application. Figure 4.1 shows the installed SSN applications of a user, the joined SSNs and which SSNs belong to which applications. For instance, application A is an application for scheduling events. SSNs 1-3 are SSNs for scheduling a business meeting, a journey with the user's family and the next poker game. Application B is a shoutwall. SSN 4 is a shoutwall at a pub. Application C is used for distributing notifications. SSNs 5-6 are used by a boss notifying her/his employees about urgent issues and by a man that participates at a presentation and sends the latest findings to all SSN members.

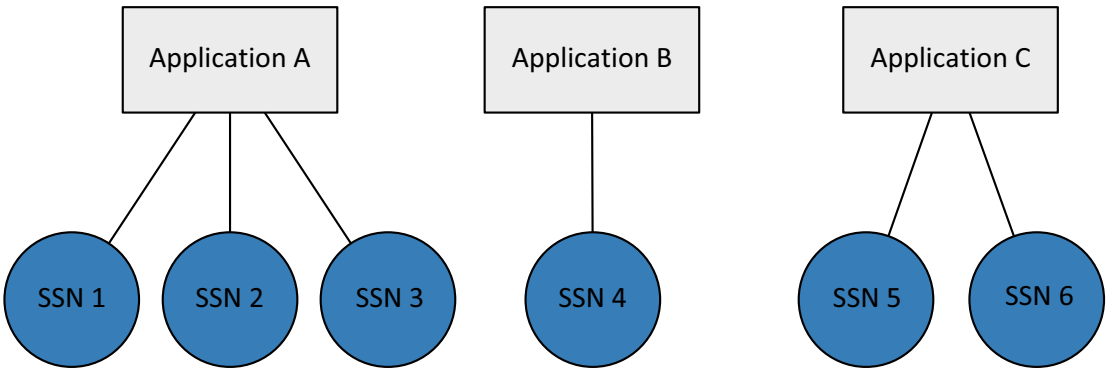


Figure 4.1: Example mapping from SSNs to applications: This figure shows the applications and SSNs of a user. Several SSNs can belong to one application.

An SSN offers one or more features such as a chat, exchanging media and creating content. A message can be nearly everything such as a text message, an image, a video, a file or other data. We decided to use XMPP for the message exchange and therefore the internet as a communication channel. XMPP solves the low-level message exchange and enables us to focus on the SSN approach and not to handle the message exchange on a per byte level. The message exchange in SSNs is based on the Publish-Subscribe paradigm that is implemented in PubSub as an extension of XMPP.

Figure 4.2 shows the lifecycle of an SSN. First of all, an SSN is created by a user (creator or owner). Other users discover the SSN, authenticate and join (subscribe) the SSN. They are able to join an SSN at anytime during the SSN's life span. Then the users are able to use the SSN. Using an SSN means to exchange messages. Depending on the purpose of the SSN, all or only some SSN members are allowed to publish and/or receive messages. A user can decide to leave (unsubscribe) an SSN. If the life time of the SSN expires, the SSN will be deleted automatically and all SSN members will be unsubscribed. The following sections describe these steps, the architecture, buddy list and user management in detail. Furthermore, the mapping of the functionality and features to XMPP and its public extensions is defined.

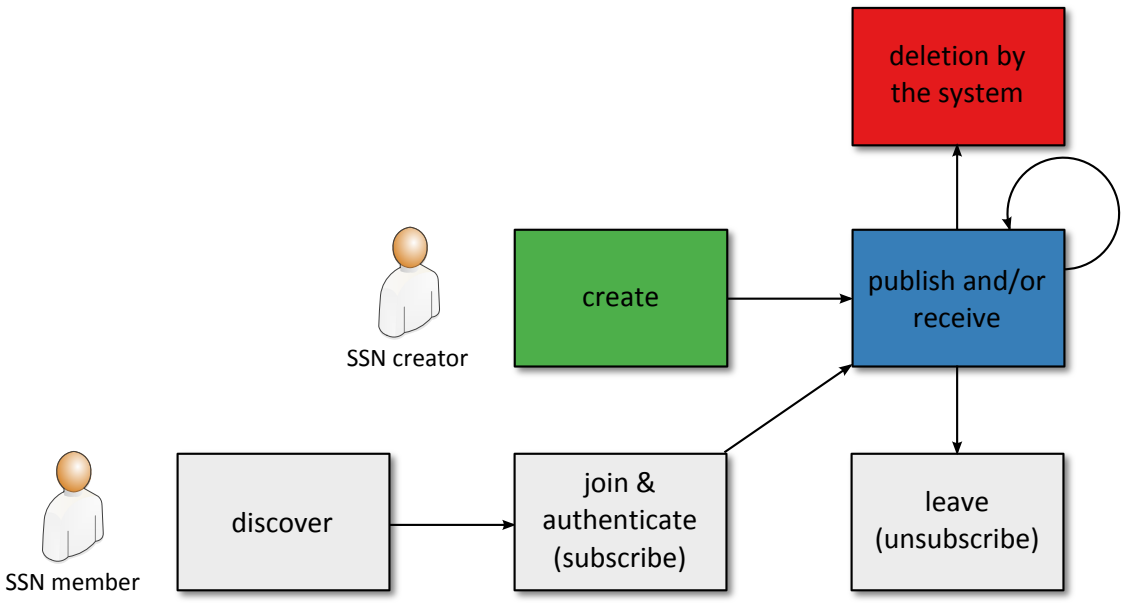


Figure 4.2: Illustration of the SSN lifecycle: The SSN is created by the SSN creator. Afterwards users are able to discover and join the SSN. The SSN members are able to exchange messages based on the properties of the SSN. A member might leave an SSN. The SSN is deleted after the time to live (TTL) has expired and all members are unsubscribed.

An SSN has several features and properties. They are specified by the SSN application or the application can ask the user to configure some or all properties of an SSN. Examples for the features and properties are:

- **Access Model:** Defines who has access to an SSN
- **Join Type:** Defines how people are assigned to SSNs
- **Time to live (TTL):** Defines how long an SSN exists
- **Communication Model:** Defines in which direction or directions the communication takes place (i.e., defines who is able to publish messages and who is able to receive messages)
- **Privacy Model:** Defines which SSN members are allowed to see the publisher's user name
- **Message Persistence:** Defines if messages should be stored on the SSN
- **Metadata:** Defines the language, title and description of an SSN

4.2 Message Exchange

As described in the fundamentals (see Section 3.4.3), the XMPP's PubSub extension [XSF10b] is based on the Publish-Subscribe paradigm. The basic concept in PubSub is a node that handles the message exchange. Such a PubSub node is used for communication in an SSN. The communication can take place in different directions. Because of that, we define the following communication models:

- **1 to N:** Only the SSN creator publishes messages to many SSN members. For instance, one person sends notifications to an interested audience.
- **N to 1:** Many SSN members publish messages to only the SSN creator (e.g., for data acquisition, sensor data is sent to a bot that displays or processes the data).
- **N to N:** Many SSN members publish messages to many other SSN members (e.g., chat, content sharing, collaborative tasks).

However, not all SSNs can be realized by only one PubSub node. An SSN is able to offer more than one feature such as voting and sending notifications from the owner to all members in a voting application. In the voting application the SSN creator sends the voting choices to all users and the users send the votes back to the creator. These two features have different properties. For instance, the communication model for the votes is N to 1 and the communication model for the notifications is 1 to N. Therefore, the voting application requires two PubSub nodes. Figure 4.3 illustrates the SSN messaging concept for an N to 1 and 1 to N communication model.

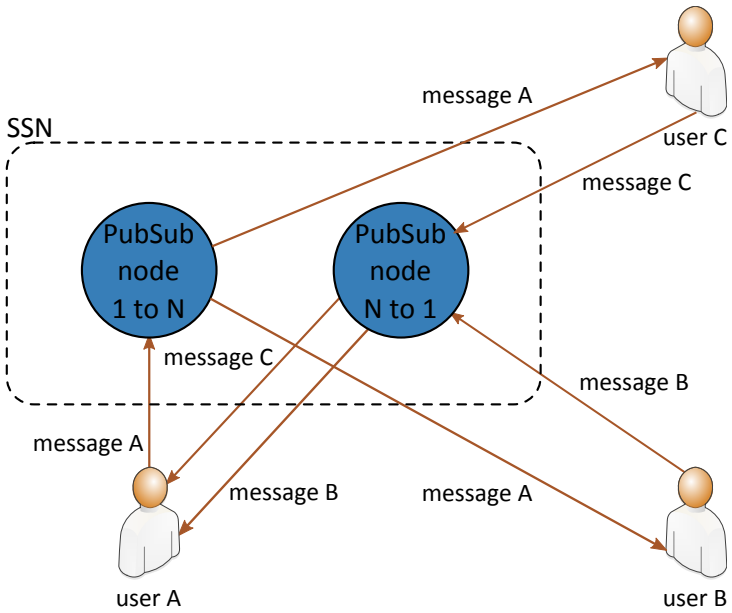


Figure 4.3: Illustration of the SSN messaging concept for an N to 1 and 1 to N communication: Message A is sent from user A to user B and user C (i.e., 1 to N communication). Message B/C is sent from user B/C to user A (i.e., N to 1 communication).

Beside the communication model, there are also other properties that are defined per node. For instance, in a voting application where all SSN members are able to suggest choices, the choices have to be stored for members that will join the SSN in the future. Furthermore, people can be disconnected for some time

due to connection losses or other user activities. Therefore, the missed messages have to be delivered to the users when reentering the SSN. We will refer to this feature as "offline message delivery" in this thesis later on.

The mapping of the communication model, message storage and offline delivery to PubSub is straightforward. The communication model affects the access model (`pubsub#access_model`) and the publish model (`pubsub#publish_model`). The message storage is also a built-in PubSub feature. However, the delivering of a large number of items (i.e., large result set) is a problem. PubSub will deliver only a subset of the items if the number of items exceeds a predefined value. This value is typically 20. XMPP defines the Result Set Management extension (XEP-0059) [XSF06b] therefore. The Result Set Management (RSM) extension enables entities to retrieve the number of items of a result set, to page forward and backward through a result set and to retrieve an arbitrary page in the result set. The offline message delivery is realized with the help of the PubSub event message type (`pubsub#notification_type`), because messages of type "normal" are stored when the user is offline and are delivered when the user comes online the next time. In contrast, a message of type "headline" is not stored for later delivery [XSF09c, XSF06a]. The metadata can also be stored directly in one of the PubSub nodes. They are not stored as PubSub items, but in the PubSub node configuration and are retrieved by a `disco#info` service discovery request.

An SSN is addressed by an automatically generated identifier containing random digits (0-9), letters (a-z A-Z) and special characters (. - ! # $ % = ( ) +). A random generated identifier prevents users from guessing SSN addresses and to protect the system from SSN address collisions. These collisions would appear if the users defined the SSN addresses. The disadvantage of the addressing approach is that the SSN addresses are not well readable for humans. Therefore, the users should not get in contact with the SSN addresses. The length of the SSN address is in a range of 6 to 10 characters. The SSN address is the identifier of the PubSub node. If an SSN consists of more than one PubSub node, the application that owns the SSN will appoint one node as the main node. The main node's ID is the SSN ID. The application must specify additional identifiers for the other nodes. An additional identifier can contain the same characters than the SSN address. The corresponding node ID consists of the additional identifier prefixed by the SSN ID and an underscore ("_") as separator.

4.3 Architecture

The SSN approach is not tied to a specific platform or programming language. SSNs can be used on PCs such as notebooks, tablet PCs or smartphones. XMPP is supported on many platforms and programming languages. Although XMPP is decentralized, the SSN architecture is centralized, because the system is controlled by a single XMPP server. Figure 4.4 shows the SSN architecture.

The XMPP server is used to enable the XMPP message exchange, to manage the users, to store the buddy list and for storing offline messages. It offers a lot of services such as the PubSub service. The PubSub service hosts the SSNs.

On the client-side there is the SSN framework and several SSN applications that use the SSN framework. The SSN framework is an own application. It enables communication in SSNs on the client-side and is capable of creating, discovering and managing SSNs. Furthermore, it provides some useful features such as a buddy list, settings and a notification system. The SSN framework consists of the SSN service, the SSN library and the XMPP library. The SSN service is the broker between the applications and the SSN framework. Each application is able to bind to the SSN service and call methods through interfaces to use the functionality of the SSN framework. The SSN library encapsulates the functionality of the SSN

framework. The XMPP library enables the communication over XMPP and bundles the functionality provided by XMPP and its extensions. The big advantage of this approach is that there is exactly one connection to the XMPP server for all SSN applications. Therefore, this approach reduces the data overhead and saves battery power. The communication between the server and the client is realized by a long-lived bidirectional TCP connection as described in the fundamentals (see Section 3.3.6) and the XMPP specification [SA04a].

The critical part when mapping this architecture to a concrete platform is the SSN service. On Android the SSN service is realized by a remote service and interprocess communication as described in Section 3.2.7. On other platforms similar techniques have to be used or if no such techniques are available they must be implemented. In the remainder of this thesis we focus on the implementation of the SSN service on Android. The SSN library is implemented in Java and therefore platform-independent. There are XMPP libraries for Java readily available.

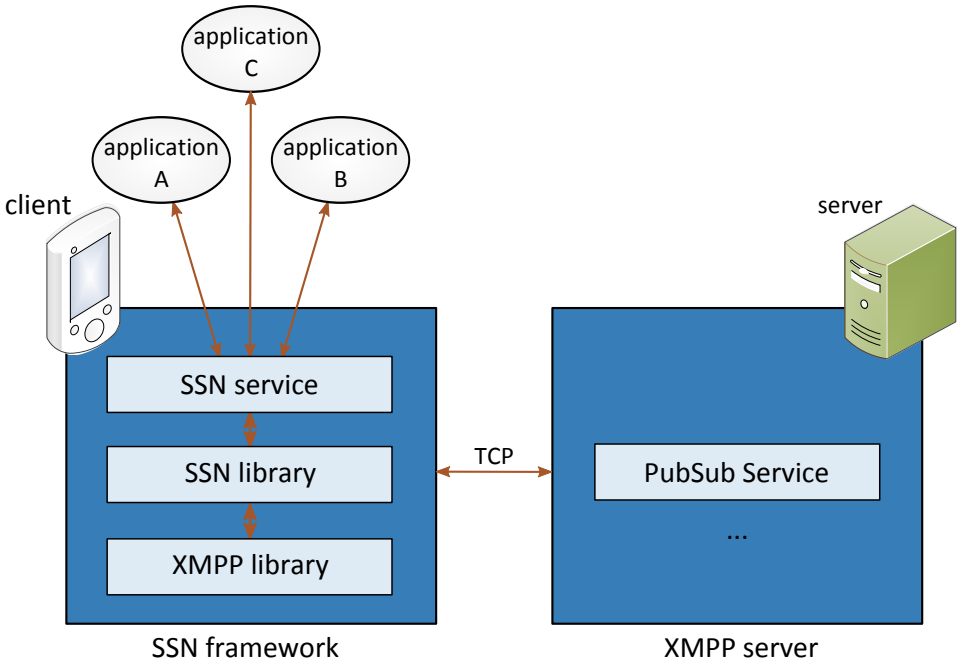


Figure 4.4: SSN architecture: The SSN framework enables communication in SSNs on the client-side. The SSN applications use the SSN framework by binding to the SSN service. On the server-side there is an XMPP server that is used to enable the XMPP message exchange. The PubSub service of the XMPP server hosts the SSNs.

4.4 User Management

Each user is identified by a unique user name. The user name is the node identifier of the JID in XMPP and is chosen by the user. The maximum number of characters in the user name is 64. The user name is case-insensitive and must not contain the following characters due to the specification of the JID [SA04a]: space " & ' / : < > @. The resource of the connection to the XMPP server should be "ssn". The users must register at the XMPP server by specifying the user name and a password to use the SSN system. The minimum number of characters in the password is 4. A user is able to change her/his password after the registration. The in-band registration extension (XEP-0077) [XSF09b] is used

therefore. It specifies a standard for the registration of users over the XMPP network, but does not define any approach against the registration of spam accounts. Bots are able to register a huge number of spam accounts in a short time. The CAPTCHA Forms extension (XEP-0158) [XSF08c] defines an enhancement for the in-band registration extension. It uses the "Completely Automated Public Turing Test to Tell Computers and Humans Apart" (CAPTCHA) to distinguish between human users and bots. CAPTCHA shows images to the user and the user must enter the shown alphanumeric characters in the registration form. The server will only accept the registration if the entered characters and digits are the same than those on the CAPTCHA image. When registering a new SSN account, the user must pass the CAPTCHA.

4.5 Buddy List

Buddy lists are commonly used in instant messaging services to establish the message exchange with other users and to see the online status of a user's contacts. The SSN buddy list typically consists of friends, colleagues and users that become acquainted with each other in a certain SSN. It is used to discover new SSNs and authenticate users in an SSN. Users are able to group related buddies into buddy groups such as co-workers, fellow students, family people that share the same interest and people that participate in the same association. The buddy list is a built-in feature of XMPP, called roster. In contrast to buddy lists in some instant messenger services, the buddy request of a user must be acknowledged by the receiver. As described in the fundamentals (see Section 3.3.3) the requester is now subscribed to the receiver's presence. Therefore, the receiver sends a buddy request to the sender, which is acknowledged automatically by the SSN framework. Two buddies have always a presence subscription of "both".

In some respects the buddy list can be seen as a bridge between SSNs or an own social network, but it is not possible to see the buddies of the buddies in the SSN system. The graph of a single SSN is a complete graph. In a complete graph each node is connected with every other node in the graph. Thus, each SSN member is one degree away from every other SSN member. The buddy list connects these complete graphs and introduces a degree of separation greater than one as shown in Figure 1.1. User C is 3 degrees of separation away from user F, although they are not in the same SSN. The users of SSN 1 are all buddies. Therefore, the creator of SSN 1 might set its access model to "buddies of the creator" during the creation of SSN 1.

4.6 SSN Discovery

4.6.1 Overview

An important part in SSNs is the discovery of new SSNs. The SSN framework has to find out the address of an unknown SSN. As mentioned, users should not get in contact with the SSN address and therefore should not be able to enter the address for discovering new SSNs. After the acquirement of the address, the SSN framework is able to retrieve the SSN's metadata. The metadata includes all information of the SSN such as SSN title, description, language, privacy model, access model, the creator's user name, the creation date, the TTL and if the SSN is buddy discoverable. From the SSN framework's perspective it is not enough to discover an SSN address, because SSNs are usually tightly coupled to SSN applications. That is, SSN application A will not understand or is able to use the SSNs of application B, because the SSN can contain any kind of data or protocol. Therefore, the name of the SSN application package is transmitted together with the SSN address.

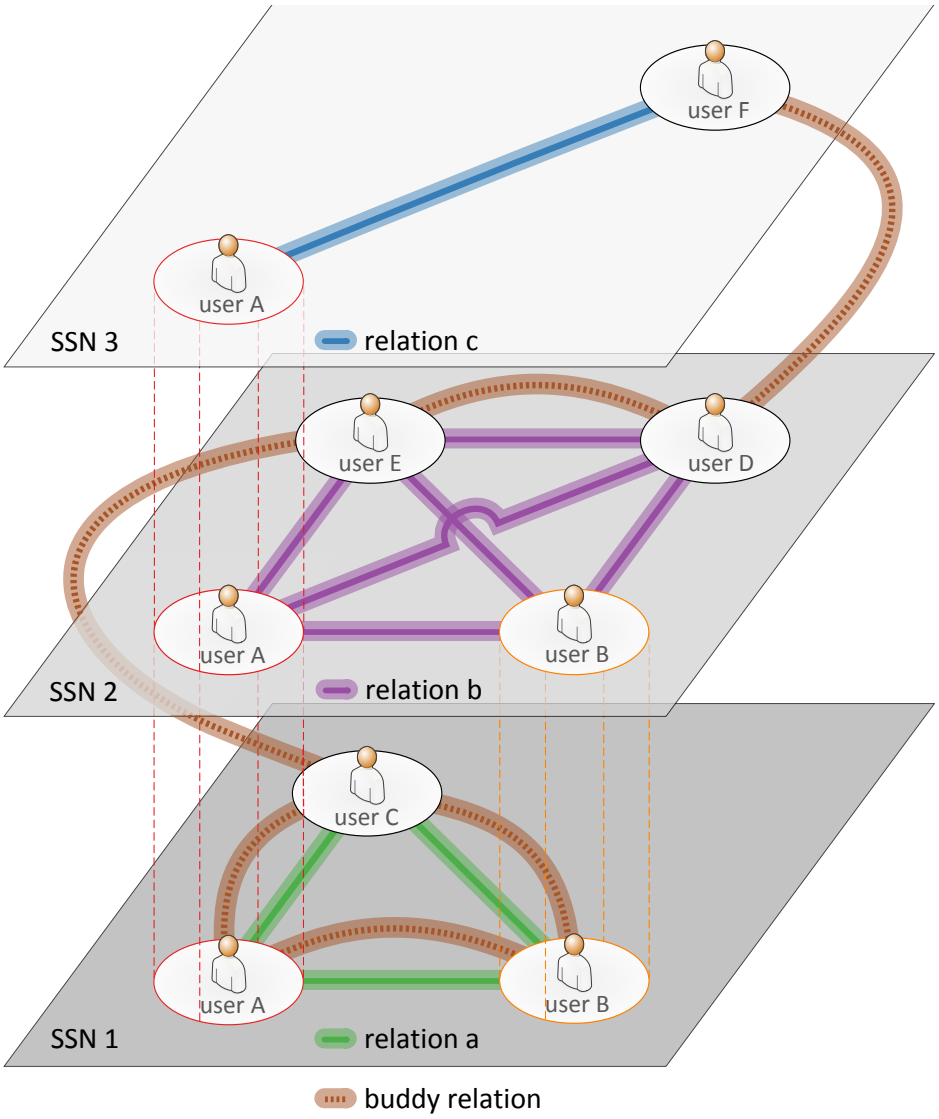


Figure 4.5: Illustration of SSNs including the buddy relation: All SSNs that are joined by user A are shown. Each of the three shown SSNs defines a different social relation among its members. Furthermore, the buddies also define a social relation. The buddy social tie of user A and user B is only plotted in SSN 1!

Obviously, there are several simple ways to distribute the SSN address such as e-mail, SMS and sharing via social network sites. However, these approaches are not very appealing and rely on external services. The discovery approaches can be classified into decentralized and centralized discovery. In decentralized discovery the address is shared among the creator and other members. It can be classified again into global and location-centric approaches. In contrast to global approaches, in location-centric approaches the SSN addresses are only distributed to colocated users. Sending invitations, retrieving the SSNs of the buddies or the mentioned sharing via e-mail, SMS and social network sites are examples for global approaches. The motivation behind the location-centric approach is a group of colocated people that intend to create an SSN for only this group. One person of this group must create the SSN. But how could the other persons become members of the SSN easily? Invitations are not the right approach, because the group members do not have to know each other and therefore do not know the others SSN user names or e-mail addresses. However, the physical proximity of the users can be used to share the

SSN address among the group members. For this purpose the SSN creator encodes the SSN address into a medium such as tones or images and the group members decode the medium back into the SSN address. Fortunately, smartphones usually provide built-in hardware support such as camera and microphone to realize such sophisticated discovery approaches. In centralized discovery approaches, a global service keeps track of all available SSNs. Users are able to ask for SSNs or are automatically assigned based on properties like location, interests, time, activity and any combination of these. The following sections describe the mentioned approaches in detail.

4.6.2 Invitation Model

The invitation discovery approach will only be used if the invitation sender is acquainted with the invitation receiver. Each member of an SSN is able to invite any other user to the SSN by sending a message containing the SSN address and the application package. The receiver of an invitation is notified of any incoming notification. This approach is very easy to realize in XMPP by sending a chat message using a suitable XML schema containing the SSN address and the application package. The XML schema is defined in Appendix C.

4.6.3 Similarity Matching

For the similarity matching discovery approach the user specifies one, all or several of the following properties: Interests, location, time and activity. Interests are some or all interests of the user such as skiing, knitting, reading or historic cars. Location is the current user location, which is usually retrieved by GPS automatically. The user can specify the size of the proximity (s)he is interested in such as 1 km. Time is a time period or frequent occurrences such as the time period for a football game or the way to work every day from 7:00 am to 8:00 am. Activity is the current activity of a user such as cooking, driving in a car or participating in an event. The used SSN application and the language if text messages are exchanged are also properties that must be specified by the user. Then the SSN framework sends a request to a global service and the service returns matching SSN addresses. The corresponding SSN metadata is shown to the user and the user decides to join the SSNs. The user is also able to receive notifications of new adequate SSNs. The SSN creator specifies none, one, all or several of the mentioned properties during the creation of an SSN.

The specified properties of the user and the specified properties of the SSN creator are compared with each other. Whereas times can be compared easily, the SSN's location has to be in the close proximity of the user's location. The proximity is defined by a default parameter or is specified by the user. The similarity of the interests and activity is determined by semantic similarity matching of the words.

The similarity matching discovery approach requires a global service and a database that stores all properties of the SSNs. Either a service on the XMPP server or a component bot is needed for this purpose. The user should not be able to store the SSN properties in the database directly, because then users would be able to advertise SSNs that do not exist (i.e., spamming).

4.6.4 Visual Codes

In SSN discovery via visual codes, the SSN address and the SSN application is encoded into a visual code. The visual code is displayed on a member's device. The user that wants to become a member of the SSN scans the visual code with the help of a camera. The visual code is decoded into the address and

application. We use QR codes [ISO00] for this purpose. "QR Code is a matrix symbology consisting of an array of nominally square modules arranged in an overall square pattern, including a unique finder pattern located at three corners of the symbol and intended to assist in easy location of its position, size and inclination." [ISO00] It is widely used for tagging components in manufacturing. Nowadays, it is used for instance in augmented reality applications and for mobile tagging. QR codes provide error correction based on a Read-Solomon code. The error correction will be useful if for example parts of the QR code are occluded, damaged or shadows or bad lighting occur. There are different versions of QR codes. Each version is able to encode a different amount of data and requires therefore a different number of square modules. Figure 4.6 shows an example of a QR code. The data area encodes the data. The alignment pattern and position detection pattern are used to scan QR codes easily regardless of the orientation of the scanner.

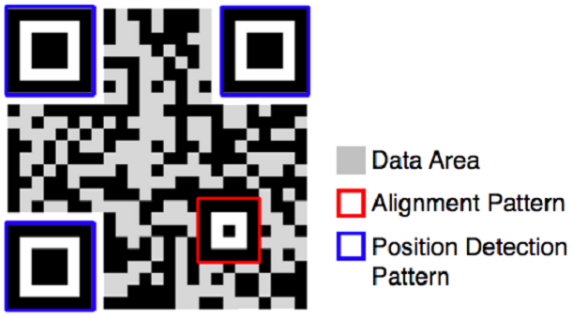


Figure 4.6: A QR code [KTC09]: Consists of the data area (square modules), alignment pattern and position detection pattern.

The discovery via visual codes can be used to discover SSNs of colocated users. It requires intervisibility to the member's device. Therefore, the member is able to control the access to the SSN. However, QR codes can also be displayed on web pages or other applications, shared via e-mail or printed out and hung up at certain locations (e.g., for events, conferences). In these cases access control can only be realized by the SSN access models.

4.6.5 Tones

The SSN discovery via tones is very similar to the discovery via visual codes except of using a sequence of tones instead of a QR code. The tones encode the SSN address and the SSN application. They are played on a member's device. The user that wants to become a member of the SSN records the tones with the help of a microphone and recognizes them. The recognized sequence of tones is decoded into the address and application.

The discovery via tones can be used to discover SSNs of colocated users. If several groups are colocated and each group has its own SSN, it will be difficult to handle that only the group members are assigned to the correct SSN. For this purpose the discovery via visual codes performs better. Of course, the tones can also be played in other applications or on web pages.

4.6.6 Buddy Based Approach

The buddy based discovery approach enables a user to discover SSNs of her/his buddies. A user is able to see the SSN applications and the SSNs of her/his buddies in the buddy list and can decide whether to join the SSN or not. The buddy based discovery approach raises a security and privacy problem. There are certain SSNs that are created for a limited group such as SSNs for event scheduling, voting, file and photo sharing. The buddies that are not in this group should not get to know of such SSNs. Therefore, a new feature for SSNs is introduced - called buddy discoverable. Non-buddy discoverable SSNs are not discoverable by the buddy based discovery approach. Furthermore, a user is able to turn off the buddy based discovery approach due to privacy reasons.

To realize the buddy based discovery approach, each user must store all SSNs and every buddy must be able to retrieve all SSNs of her/his buddies. The SSN framework has to store all joined SSNs anyway to keep track of them. The Private Eventing Protocol (PEP) [XSF10c] is used for this purpose. PEP is introduced in Section 3.4.4. It is basically a user-oriented subset of PubSub and can be used to store private, public or semi-public data [XSF08d, XSF08e]. The advantage of PEP is that the stored data of a user is retrievable by certain other users even if the user is offline. There is one PEP node for all installed SSN applications, and two PEP nodes are created per application. One PEP node is used for the buddy based discovery approach. It has an access model of "presence" to restrict the access to the user and buddies of the user. The other PEP node is used for non-buddy discoverable SSNs and therefore has an access model of "whitelist". If a user turns off the buddy based discovery approach, the access model of all created PEP nodes will be set to "whitelist".

4.7 Security and Privacy

The scenarios in the motivation section raise different requirements and problems. One of these problems is that a certain group (e.g., friends) wants to create an SSN where only this group has access. The random addressing approach secures the SSNs, because the address is very difficult to guess. However, the discovery approaches - especially the centralized and global discovery approaches - negate this advantage. Therefore, the access model is introduced as a feature for SSNs. The access model restricts the access of users to SSNs. There are a lot of access models conceivable. Access models such as "on request" (i.e., the user sends a subscription request and someone must permit or deny access) will require manual intervention of the SSN owner or all members. Whereas the SSN owner is not online at any time, the broadcasting of the authentication request causes a lot of messages and the users will become frustrated. Because of that, this kind of access models is not suitable for SSNs. Thus, there are only the following access models.

- **Open:** Everybody is allowed to join the SSN.
- **Buddies of the creator:** Only all buddies or certain groups of the SSN creator's buddy list are allowed to join the SSN.

To emphasize the spontaneity of SSNs, an SSN has only a limited time to live (TTL). The minimum TTL is 10 minutes and the maximum TTL is 7 days from the creation date onwards and is specified when creating an SSN. This design decision improves privacy and protects data significantly, because the SSN data is not stored on the XMPP server forever.

In SSNs not each message must include the sender's user name. SSNs where the sender's user name is not included in the messages are called anonymous SSNs. For instance, an anonymous SSN can be useful to protect the privacy in SSNs that collect sensitive data and can therefore be used for anonymous data acquisition. The following privacy models are defined:

- **Anonymous:** Nobody receives the sender's user name.
- **Semi-anonymous:** Only the SSN creator receives the sender's user name.
- **Public:** Every member receives the sender's user name. In SSNs with an open access model every user that discovered the SSN is able to read the messages and therefore assign the messages to the corresponding publishers.

Whereas the mapping of the access model to PubSub is straightforward (PubSub access models: open, roster, presence), the TTL for nodes is not specified in PubSub. There is only a TTL for PubSub items (`pubsub#item_expire`). Therefore, the (`pubsub#node_expire`) configuration option is introduced, which specifies the TTL of a PubSub node in seconds. Furthermore, the sender's JID of a PubSub message is usually not included in the item. However, the PubSub extension specifies the `pubsub#itemreply` option where the publisher's user name is provided in the item. The node owner is able to specify who is able to receive the publisher's name of the item, but only the affiliations "owner" or "publisher" are allowed. However, only the owner is able to manage affiliations. Thus, the owner would have to assign each new subscriber the publisher affiliation. This is impossible to handle, because this would again require that the owner is always online. Therefore, some values are added to the `pubsub#itemreply` option. The value "none" (named after the affiliation) is used for public SSNs, "nobody" for anonymous SSNs and "owner" for semi-anonymous SSNs. All the mentioned changes are documented in Appendix D.

There are some further issues concerning XMPP. Each user knows the node identifier of the JID and the password. The domain identifier of the JID and the PubSub node identifiers are not difficult to achieve. Therefore, each user is able to use her/his own XMPP client or XMPP instant messenger client that allows users to send any stanzas to use and manipulate SSNs. The SSN creator is the node owner of the used PubSub nodes. The node owner in PubSub is the administrator of the PubSub node and is therefore able to do nearly everything (s)he wants. For instance, the owner is able to modify the configuration, manage affiliations and subscriptions of the node, delete the node, delete items of all publishers from the node and purge the node. The modification of the PubSub node configuration may violate the system's consistency, security and privacy. For example, the access model and the privacy model can be modified without the members' notice. The management of the affiliations/subscriptions may also violate the system's consistence and users may be banned/unsubscribed from an SSN. These issues occur only in PubSub and not in PEP. Therefore, the node configuration, the management of affiliations, the node deletion, purging nodes and the management of subscriptions are disabled in PubSub. The deletion of items is restricted only to the publisher of the item. All these changes are documented in Appendix D.

The user's online status is shared with her/his buddies. A user is able to prohibit this. The Privacy Lists extension (XEP-0016) [XSF07], which is described in Section 3.4.2, is used for this purpose. If a user decides to prohibit the sharing of her/his online status, the sending of presence messages of type "both" and "from" is denied. Furthermore, some users might not want to use the buddy list and are therefore able to configure the SSN framework to automatically deny all future buddy requests.

In SSNs not all spam messages (spim) are visible to the users, because the XMPP message could contain arbitrary XML. The receiver may not be able to parse the message's XML and therefore the message is

not displayed to the end user. That is, a user may receive a huge number of messages without notice, which drains the battery, costs bandwidth, data volume and processing power and may delay important messages. The size of a message is usually limited by the XMPP server implementation. Commonly used limits are for instance 64 kB. However, 1000 messages with a size of 64 kB add up to 64 MB. The Privacy Lists extension [XSF07] is used to reduce spam in SSNs. The incoming messages of type "message" are restricted to messages from buddies, which are used for invitations, and messages from the PubSub service. However, spam within SSNs cannot be handled with privacy lists, because the publisher of the PubSub item is not the sender of the message (the sender of the message is always the PubSub service). Currently there is no simple solution to this problem. Though, a user could be warned if an SSN generates a huge number of messages/traffic and the user could be advised to leave the SSN.

Chapter 5

Implementation

5.1 Overview

As already mentioned, SSNs are not tied to any platform or programming language. However, in our reference implementation we focus on mobile devices. The SSN service is implemented on Android. The SSN library is implemented in Java and is therefore platform-independent. Thus, it is possible to implement bots on top of the SSN library.

The SSN framework is an own Android application. It consists in fact of two subapplications, which are bundled in a single Android application (i.e., one apk) file. Each of these subapplications runs in its own process. The first application - we will refer to this application as SSN administration application - offers users interfaces for registration, SSN discovery, a buddy list, settings, the notification system, etc. The second application is the SSN Service. It is realized by an Android remote service, which is described in Section 3.2.4. The SSN service encapsulates all functionality used in SSNs. It is responsible for the client-side communication in SSNs and uses the XMPP library Asmack. The SSN administration application and all SSN applications bind to the SSN service and call methods in an interface to use the functionality of the SSN framework such as publishing to an SSN. registering a listener for an SSN. unsubscribe from an SSN and get the stored items of an SSN. Interprocess communication as described in Section 3.2.7 has to be used, because the SSN service and the SSN applications run in different processes.

The SSN framework offers many features. Some of these features (activities) can be launched by the SSN applications such as the activity for discovering new SSNs, the buddy list and the notification list. The main features are:

- **User management:** Registration of a new user (see Figure 5.1 (a)), login and password change
- **Buddy List:** Enables the user to see the online status, the joined SSNs and the installed SSN applications of her/his buddies (see Figure 5.2 (a) and Figure 5.5 (c))
- **Notification System:** Manages and displays notifications of new SSN invitations and buddy requests; moreover, each application is able to create notifications (see Figure 5.2 (b))
- **Settings:** The user is able to configure the SSN framework such as settings for the connection (e.g., proxy and reconnection delays), privacy (hide online status, decline all buddy requests and

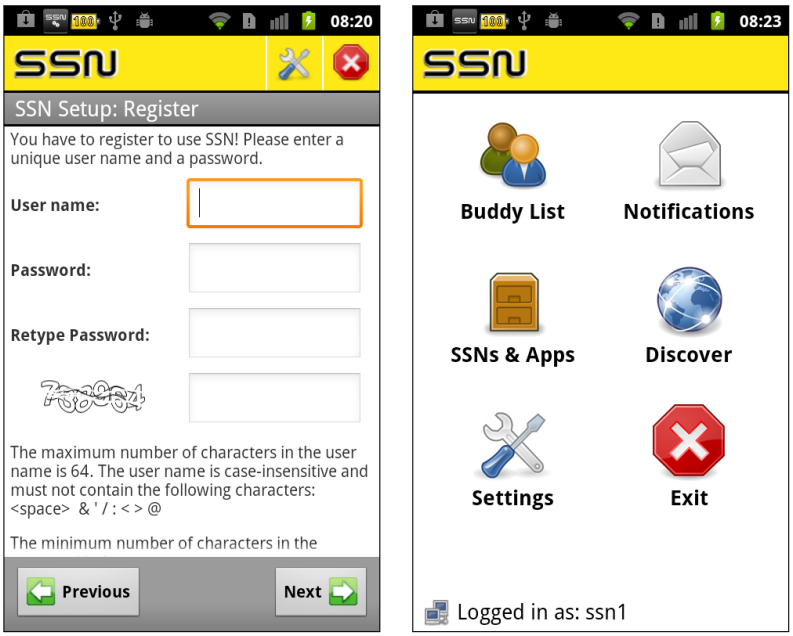
enable/disable buddy based discovery) and notifications (e.g., notification type: sound, vibration, (flashing) lights and connection status). (see Figure 5.2 (c))

- **SSN Administration:** Creates new SSNs and manage joined SSNs by the SSN and application list (see Figure 5.3 (a))
- **SSN Application Management:** Keeps track of all installed SSN applications; enables to share the SSN application name to other users
- **SSN Sharing:** Shares joined SSNs, which can be discovered by other users (see Figure 5.4)
- **SSN Discovery:** Discovers new SSNs (see Section 5.3 and Figure 5.5)
- **Connection Management:** Creates and maintains the connection to the XMPP server; reconnects after connection losses

For easier understanding, a typical user's work flow of the SSN implementation is described. After the installation of the SSN framework from the Android Market, the SSN setup is launched. The user must register with a user name and password at the system as shown in Figure 5.1 (a). Then the user is able to use the features of the SSN framework as shown in Figure 5.1 (b). If the user discovers an SSN. (s)he will be prompted to install the required SSN application from the Android Market. The user is also able to search and install new SSN applications without discovering an SSN. The user is able to create new SSNs with the help of an SSN application. Depending on the application the user must specify some properties such as title, description and access model. The SSN application is able to define default values for the configuration or configure some or all properties by itself. After the successful creation of the SSN, it is displayed in the SSN and application list in the SSN framework. The SSN and application list is shown in Figure 5.3 (a). Applications might also choose to display the SSN list. The user is able to share the SSN (and the corresponding SSN application) by sending invitations, displaying the SSN's QR code or playing the SSN tones as shown in Figure 5.3 (b) and Figure 5.5. Furthermore, only the SSN application can also be shared via invitations or a QR code. The SSN framework or often the application itself enables a user to discover new SSNs by scanning a QR code, recording SSN tones, displaying the SSNs of her/his buddies and receiving SSN invitations as shown in Figure 5.3 (c) and Figure 5.4. Except the discovery via invitations, each discovery approach must be started by the user. After the discovery the SSN title, description, language, privacy model, access model, application, the creator's user name, the creation and deletion date and if the SSN is buddy discoverable are displayed and the user must confirm the SSN join as shown in Figure 5.6. The user is now able to use all discovered and created SSNs by using the corresponding SSN applications. After the SSN's time to live has expired, the SSN is removed from the SSN list automatically. However, some applications might provide a history of previously joined SSNs.

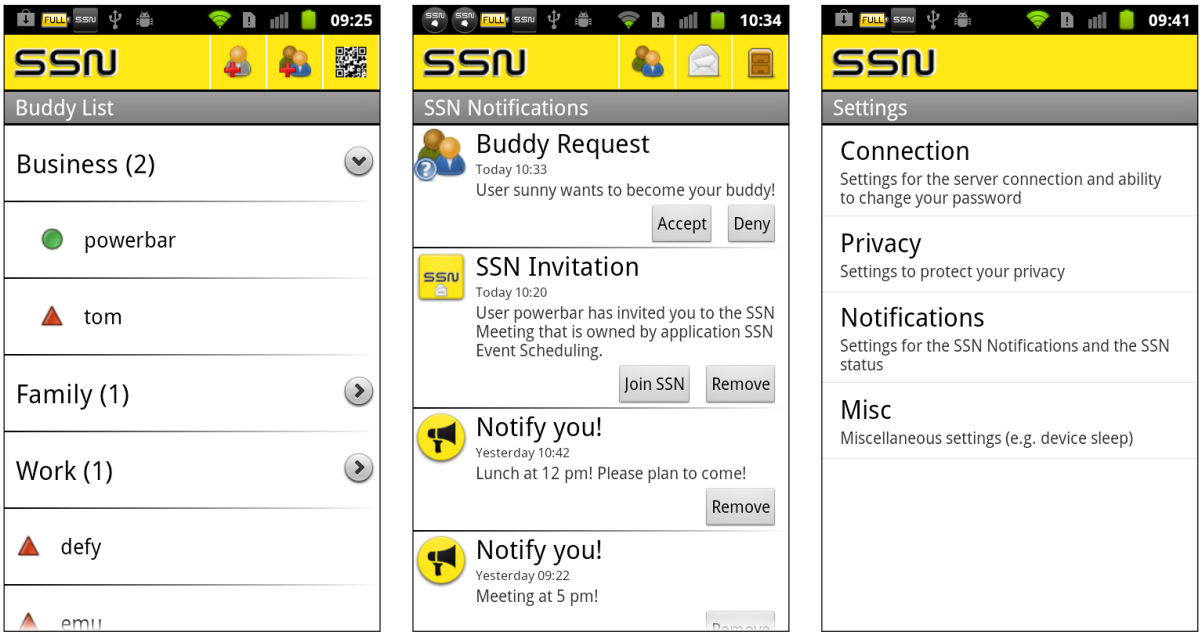
We used the dashboard, action bar and quick actions that were introduced at the Google IO 2010 [NBF+11] in our prototype implementation of the SSN framework and the implemented example applications. For instance, Figure 5.1 (b) shows a dashboard, Figure 5.2 (a) shows the action bar and Figure 5.6 (c) shows quick actions. The dashboard highlights the application's most used features and capabilities. The action bar is located at the top of a screen (activity) and replaces Android's title bar. The application logo is displayed on the left. If the user clicks on it, the home activity (dashboard) of the application is launched. Up to three icons are displayed on the right of the action bar. The icons represent common actions of the activity such as search and refresh or are used for navigation. For instance, in Figure 5.2 (a) the actions are: "Add a buddy", "create a buddy group", "show the user identity (user name) as a QR code". The quick actions are used for contextual actions. For instance, in Figure 5.6 (c) quick actions will be displayed if the user clicks on the application name (shown as a "link"). The quick actions are "description"

(i.e, show the description/help of the SSN application), "system info" (i.e., launch the package manager) and "Android market" (i.e., launch the application's detail activity on the Android market).



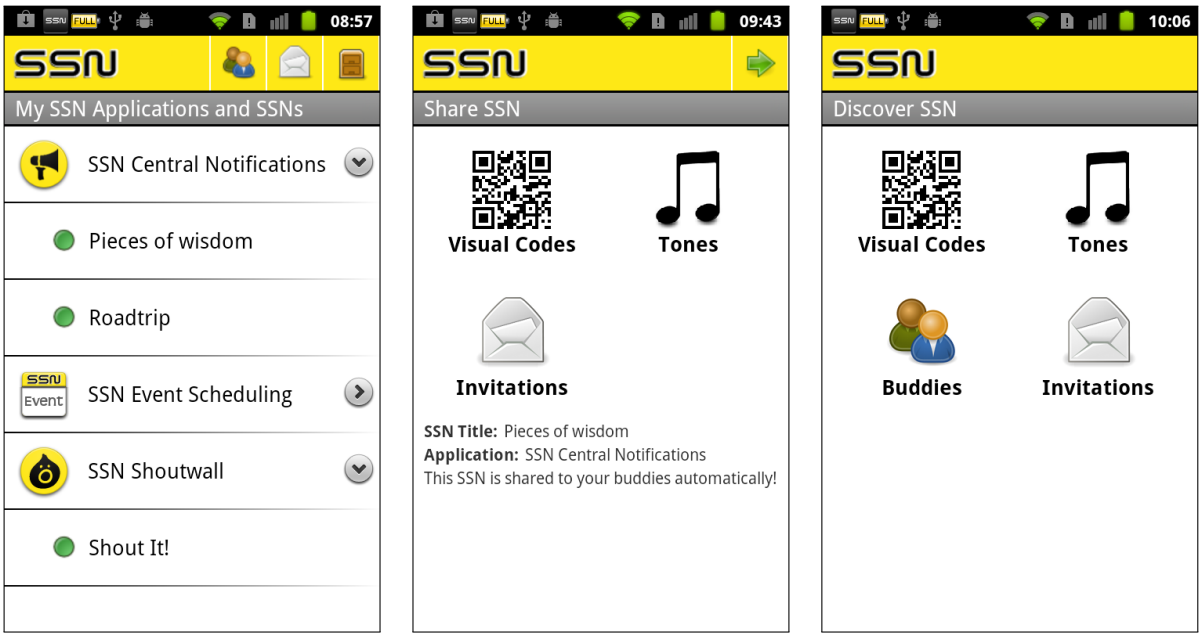
(a) (b)

Figure 5.1: The activity for registering a new user of the SSN setup (a) and the dashboard of the SSN administration application (b): In (a) the user must register with a user name, a password and (s)he must pass the CAPTCHA. The dashboard (b) shows the most important features of the SSN administration application.



(a) (b) (c)

Figure 5.2: The buddy list (a), notification list (b) and settings (c) of the SSN administration application: In the buddy list three buddy groups are shown. The green/red icon next to a buddy indicates that the buddy is online/offline. The buddy list activity also shows an action bar. In the notification list notifications of the SSN administration application (e.g., buddy request and SSN invitation) and notifications of the SSN applications are shown. The user is also notified by notifications on the status bar.

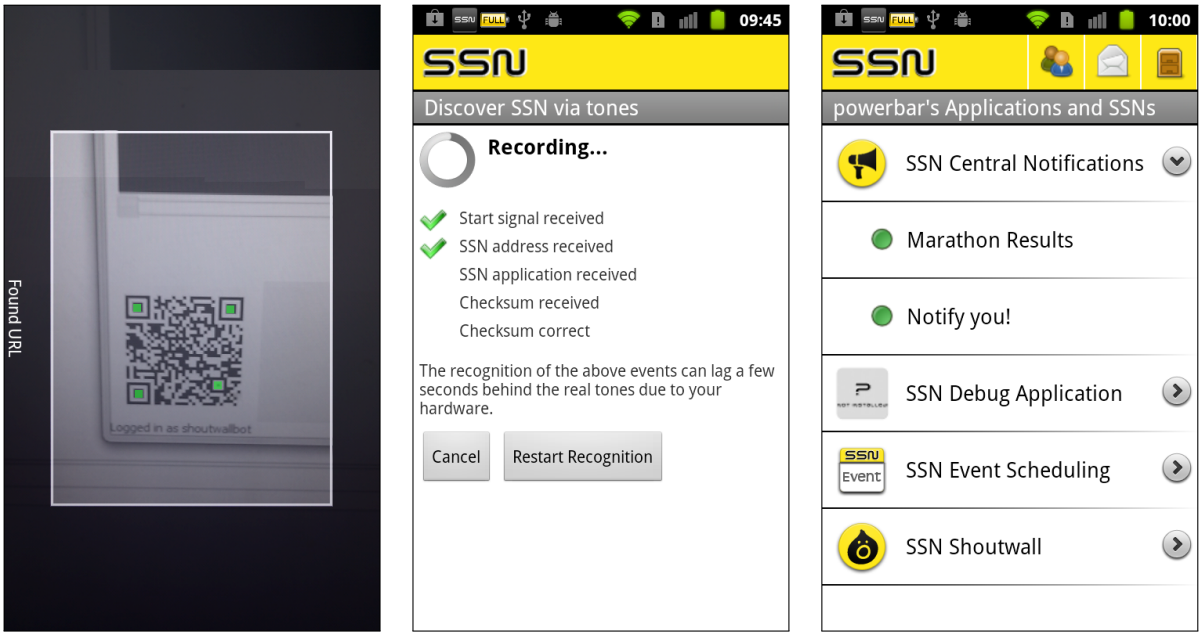


(a) (b) (c)

Figure 5.3: The SSN and application list (a) and the activities for sharing an SSN (b) and discovering an SSN (c): The icons next to the SSNs in the SSN and application list indicate the remaining TTL (i.e., an orange icon indicates if the remaining TTL is shorter than one fourth of the TTL). The activity to share SSNs can be launched for instance from the SSN and application list, the SSN applications and it might be displayed after creating a new SSN.

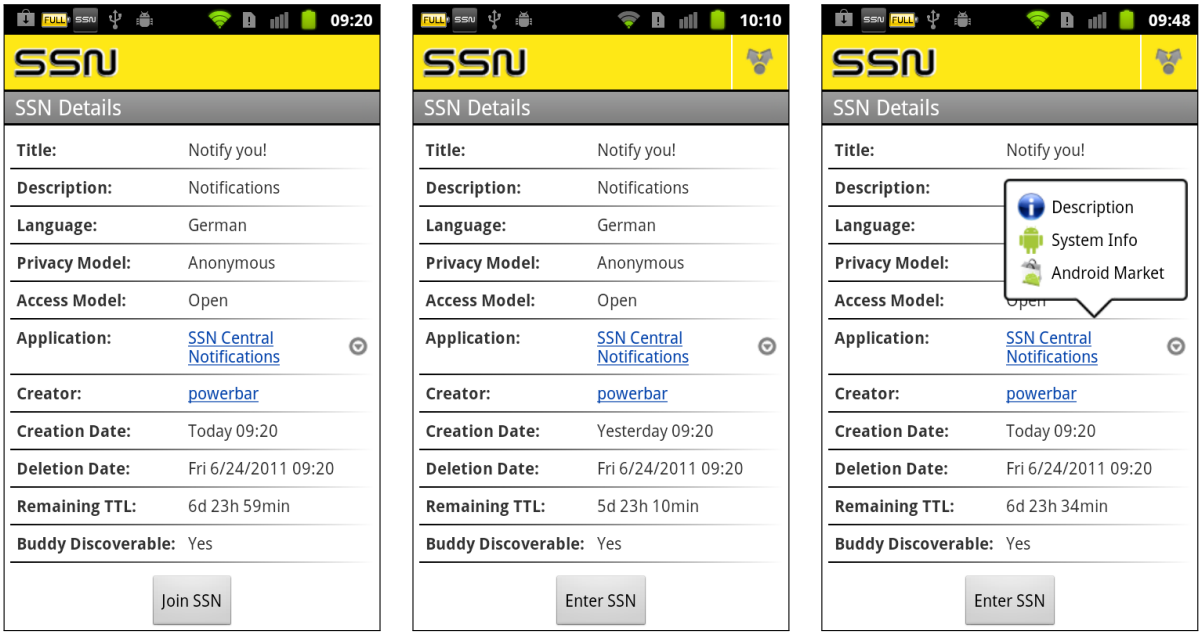
(a) (b) (c)

Figure 5.4: The activities for sharing an SSN with the help of invitations (a), visual codes (b) and tones (c): The user must select the buddies that should be invited to an SSN (a). (b) shows the activity of the Barcode Scanner from the ZXing Team [Zxi11] that displays the QR Code to share an SSN. In (c) the playing of the SSN tones can be started, stopped and restarted.



(a) (b) (c)

Figure 5.5: The activities for discovering an SSN with the help of visual codes (a), tones (b) and buddies (c): In (a) a QR Code of the YouTube Shoutwall is scanned with the help of the Barcode Scanner from the ZXing Team [Zxi11]. (b) shows the recording of the SSN tones. All recognition steps are visualized. (c) shows the SSN applications and SSNs of a buddy. The user has not installed the "SSN Debug Application". After discovering an SSN the SSN details activity is launched.



(a) (b) (c)

Figure 5.6: The SSN detail activity before joining an SSN (a) and when the SSN has already been joined ((b) and (c)). (a) is shown after discovering an SSN. (b) can be launched for instance from the SSN and application list. The user is able to share the joined SSN in (b). In (c) the quick actions when clicking on the SSN application "link" are shown.

As mentioned, each SSN has different properties. The properties are defined by an SSN application, but the SSN application can ask the user to configure some or all properties of an SSN. The SSN properties of our reference implementation are:

- **Title:** The title of the SSN
- **Description:** A short description of the SSN (e.g., purpose of the SSN)
- **Language:** The spoken language of the members; not defined if only data without text or speech is exchanged
- **Privacy Model:** Defines which users are allowed to see the publisher's name (see Section 4.7 for the available models)
- **Access Model:** Defines who has access to an SSN (see Section 4.7 for the available models)
- **Time to live (TTL):** Defines how long an SSN should exist
- **Buddy discoverable:** Defines if the SSN can be discovered via the buddy based discovery approach

The following properties are defined per SSN node (= PubSub node):

- **Protocol:** Defines the protocol of the exchanged messages
- **Communication Model:** Defines in which direction or directions the communication take place (i.e., defines who is able to publish messages and who is able to receive messages - see Section 4.2)
- **Maximum number of messages to store:** Defines if and how many messages should be stored on the node
- **Deliver messages to offline users:** Defines if messages of offline users are stored and delivered the next time the user comes online

5.2 Message Exchange

The SSN framework enables the SSN applications to send and receive messages. The messages can be of any XML scheme, but must have a unique XML namespace for the XML parsing. In XMPP, messages are not polled from the receiver, but pushed to the receiver. Asmack offers a listener to filter messages of a PubSub node. The SSN framework enables SSN applications to register listeners for SSN nodes. These listeners are set from an application component (e.g., activity, service or broadcast receiver) of the SSN application by binding to the SSN service and calling a method in an AIDL interface on the SSN service. The SSN framework receives messages of a PubSub node when there is a connection to the XMPP server, regardless of the availability of a listening SSN application. However, the received messages of inactive applications must not be discarded. Instead, the received messages are stored in an SQLite database. There are two kinds of listeners. An SSN application sets a persistent listener for an SSN node after the join or creation of an SSN. A persistent listener enables the SSN framework to filter the incoming SSN messages, but these messages are not delivered to the SSN applications. The SSN application appends a volatile listener to a persistent listener for receiving the messages of an SSN node. If there is a persistent listener set for an SSN node, but there are no volatile listeners appended, the

SSN framework will store the received message for later delivery. Whereas persistent listeners are set once, volatile listeners are set and removed frequently during the application's work flow. There is also a special kind of persistent listeners for SSN nodes that displays the received messages directly in the notification system without delivering the received message to the application. These messages must be of a predefined format and contain the application package name, the message and the buttons to display. The buttons trigger an intent, which is also specified in the message. The intent can launch an activity of the SSN application or any other installed application such as the web browser, the email application or the SMS application. The XML schema of the notifications is shown in Appendix C. However, due to the limited amount of time, in the reference implementation the buttons that trigger intents are only implemented for local usage via IPC and cannot be created remotely over XMPP, because this feature is not used by any example application.

5.3 SSN Discovery

5.3.1 Overview

The user is able to discover SSNs from the SSN administration application. An SSN application might choose to display a button for discovering an SSN from the SSN application by starting an activity from the SSN administration application. After the discovery of the SSN, the user must acknowledge the SSN join, because otherwise an SSN application would be able to join SSNs on behalf of the user without user notice. Each SSN might consist of more than one PubSub node and the SSN framework does not know the IDs of the PubSub nodes except the main node. Therefore, the SSN applications are responsible for joining new SSNs. That is, the SSN administration application sends an intent to the SSN application that is responsible for the SSN. The SSN application announces an intent filter therefore and the SSN framework uses the hybrid approach discussed in Section 3.2.4 to address the correct activity. The SSN application subscribes to all required PubSub nodes and sets the node listeners by calling the corresponding methods on the SSN service via IPC.

We implemented the discovery of SSNs with the help of invitations, via visual codes, via tones and the buddy based approach. Due to the limited amount of time, we do not implement the SSN discovery approach by similarity matching. Whereas the implementation of the on invitation and buddy based approach is straightforward (see design Sections 4.6.2 and 4.6.6), the discovery via visual codes and via tones is described in detail in this section.

5.3.2 Visual Codes Discovery

For the SSN discovery via visual codes, QR codes are used. Figure 5.7 shows an example of a QR code for a specific SSN. The Android application "Barcode Scanner" from the ZXing Team [Zxi11] is used to scan and generate QR codes. An intent activates the Barcode Scanner application. The data that is displayed as QR code is an URI of the following format:
`ssn://subscribe?app=<package name of the application>&ssn=<ssn address>`.

5.3.3 Tones Discovery

For the SSN discovery via tones the SSN address and the SSN application's package name are encoded as a sequence of sinus tones. This sequence of tones is played on the sender's device and recognized

Figure 5.7: A QR code that encodes an SSN address and an SSN application

on the receiver's device. Each sinus tone has a certain frequency. First of all, we want to describe the recognition approach. Then the mapping from symbols to frequencies and the message format are specified.

On the receiver's device the sequence of tones is recorded with the help of the microphone. The sampled audio data is decomposed into blocks and a windowing function is applied. Then the power spectrum is determined for each block by the Fast Fourier Transform (FFT) [Bri88]. The dominant frequency in the power spectrum is treated as the frequency of the tone in the processed block. The recording and recognition process is stopped after the stop tone has been recognized. Then the received frequencies are converted into the symbols and the SSN address and the SSN application's package are extracted. Finally, the received data is verified by a checksum. The following paragraphs present more detailed information on the recognition approach.

We choose a good tradeoff between accuracy and performance for the recognition approach, due to the limited processing power of mobile devices. We choose a sampling rate of 8 kHz, whereas the maximum detectable frequency is 4 kHz due to the sampling theorem. The recorded audio data is retrieved consecutively as 16 bit PCM samples and stored in a buffer. The minimum size of the buffer depends on the hardware and the drivers. Before processing each block with the FFT, the samples of the block are multiplied with a Hann window to prevent leakage. The blocks do not overlap due to performance issues. The FFT converts the discrete spatial signal into the frequency domain. It computes the Discrete Fast Fourier Transform (DFT), which is a well known and fast algorithm for the computation of the Fourier transform. "The essence of the Fourier transform of a waveform is to decompose or separate the waveform into a sum of sinusoids of different frequencies." [Bri88] The result of the FFT are complex numbers. The signal in the frequency domain (spectrum) is also discrete and therefore has limited frequency resolution. Thus, the spectrum consists of a number of bars, called bins. The represented frequencies of one bin is determined by $\frac{f_S}{N_F}$, where f_S is the sampling frequency and N_F is the number of input samples for the FFT. N_F must be a power of two. The FFT runs in $\mathcal{O}(N_F log N_F)$ time. We choose 256 for N_F, because lower values would decrease the frequency resolution and higher values would increase the runtime. The width of a bin in the spectrum is therefore 31.25 Hz. The frequency of the block is the dominant frequency in the power spectrum. The power spectrum shows the distribution of the signal's power among frequencies. It is computed by calculating the magnitude of the resulting complex numbers after the FFT. The dominant frequency is the bin with the highest value in the power spectrum. Equation 5.1 shows the formula for calculating the frequency of a bin

$$f = \frac{f_S \cdot i}{N_F}, \tag{5.1}$$

where f_S is the sampling frequency, N_F is the number of input samples for the FFT and i is the index

of the bin with the highest value in the power spectrum in the interval of $\left[0, \frac{N_F}{2}\right)$. Only the frequencies in the range of the played tones' frequencies are potential values for the dominant frequency. Figure 5.8 displays the power spectrum of several signals. (a) shows the full spectrum of a recorded 600 Hz tone with minimal background noise. The power spectrum is symmetric around the Nyquist frequency of 4 kHz. In (b) the power spectrum of the same signal as in (a) is shown in the frequency range of our alphabet. Figure 5.8 (c) and (d) displays the power spectrum of a tone with 1 kHz while playing music in the background. In (d) the bins of approximately 600 Hz and 1480 Hz are nearly as high as the bin of 1 kHz.

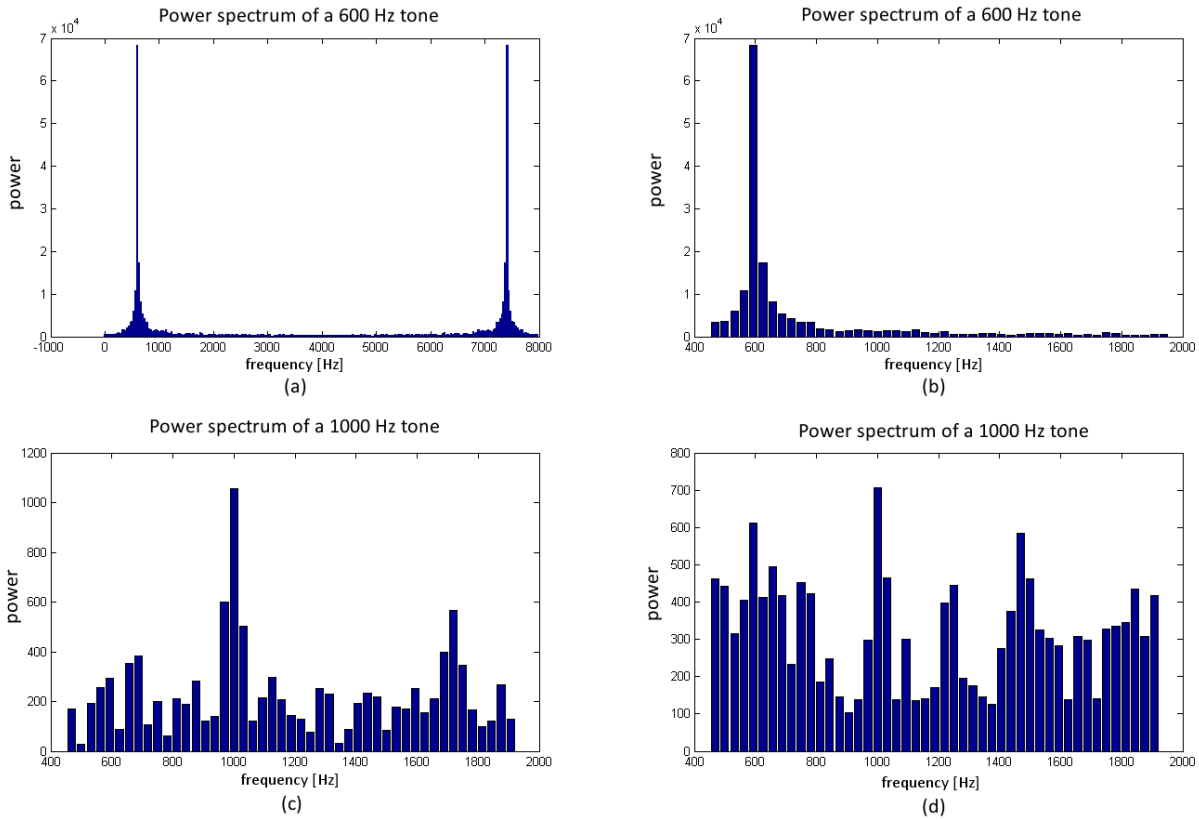


Figure 5.8: The power spectrum of several signals: (a) full spectrum of a recorded 600 Hz tone with minimal background noise; (b) power spectrum of (a) in the frequency range of our alphabet; (c) and (d) power spectrum of a recorded 1 kHz tone while playing music

Due to noise, processing errors, the windowing and hardware imperfections, recognition errors are likely. Therefore, a few blocks are processed to detect a single tone. However, this increases the duration for a single tone. We assume that two consecutively recognized frequencies of the same value are required to detect a single tone. The maximum number of recognizable frequencies per tone N_T is assumed as four. Equation 5.2 shows the formula for calculating the duration of one tone

$$l_T = \frac{N_T \cdot N_F}{f_S}, \tag{5.2}$$

where N_F is the number of input samples for the FFT and f_S is the sampling frequency. In our case l_T is 128 ms (7.8 tones per second). The detection of two consecutive played identical frequencies is a problem. Therefore, the time between two tone detections must be greater than N_T.

The recording and the recognition must be realized in two different threads, due to the following reasons. The minimum buffer size for recording varies from device to device and is not necessarily a power of two, but the input data size of the FFT must be a power of two. The second reason is the limited processing speed on mobile devices. Whereas the recognition process including the FFT and the recording are very costly, only the recording is time-critical. Therefore, the priority of the recorder thread must be much greater than the priority of the recognition thread. This results in a more frequent execution of the recorder thread in respect to the recognition thread.

Next, we define the frequency of the tone for each symbol in the used alphabet. Only a frequency range from approximately 500 Hz to 1900 Hz is used, because frequencies out of this range are annoying for most people. The parameters of the recognition approach affect the mapping from frequencies to symbols. With the mentioned parameters we have a frequency resolution of only 31.25 Hz. Deviations due to processing errors, hardware, noise, etc. are likely. We assume that the recognition error is smaller than two times the frequency resolution, which results in a feasible number of representable frequencies. Thus, the recognition interval has a length of 62.5 Hz. The frequency of the played tone is in the middle of the recognition interval. According to the Java specification [GJSB05] the Java package name contains only letters (a-z A-Z), digits (0-9) and special characters (_ .). The SSN address contains only digits (0-9), letters (a-z A-Z) and special characters (. - ! # $ % = ( ) +). Therefore, 73 symbols have to be encoded with the 24 available frequencies. Therefore, some symbols must be represented by two tones with different frequencies. Table 5.1 shows the mapping from frequencies to symbols. The letters "a" to "r" and the dot are represented by only one tone. For all other symbols two tones are needed. For a symbol in the "Symbol 2" column a tone with a frequency of 1703.125 Hz is played first. For a symbol in the "Symbol 3" column a tone with the frequency of 1765.625 Hz is played first. And for a symbol in the "Symbol 4" column a tone with the frequency of 1828.125 Hz is played first. For instance, the digit "6" is represented by a tone with a frequency of 1703.125 Hz and a tone of frequency 1390.625 Hz, whereas the letter "X" is represented by a tone of frequency 1828.125 Hz and a tone of frequency 828.125 Hz. The start, stop and separator tones are represented by a frequency of 1890.625 Hz. The number next to the character in Table 5.1 is used for calculating the checksum.

The played tones have a certain message format, which is shown in Figure 5.9. The start, stop and separator tones have the same frequency. The length of the tones representing the address / package name is usually greater than the number of characters in the address / package name. We use a simple checksum similar to ISBN-13 [ISO05] to verify the correctness of the received SSN address and package name. Each corresponding number of the character (see Table 5.1) is multiplied by the index of the character and the result of all multiplications is summed up. Finally this sum is divided by 73 to reduce the values to the length of our alphabet. The remainder is the checksum. It is converted back into a symbol. Equation 5.3 shows the formula for calculating the checksum

$$CS = \left(\sum_{i=1}^{N} x_i \cdot i \right) \mod 73, \tag{5.3}$$

where N is the number of characters in the concatenated string consisting of the SSN address and package name of the SSN application and x_i are the converted numbers of the string. Obviously, this simple checksum is not able to detect all transmission errors.

On the sender's device the SSN address and the SSN application's package name are converted into frequencies according to Table 5.1. Then the checksum is calculated, the message is assembled according to Figure 5.9 and finally the sinus tones are played. Tones with a low frequency are played louder than tones with a high frequency, because high frequencies are perceived louder by humans. The tone sequence's length depends on the characters in the transmitted string. For instance, an SSN address

Played Frequency [Hz]	Detected Frequencies [Hz]	Symbol 1		Symbol 2		Symbol 3		Symbol 4	
		Char	No.	Char	No.	Char	No.	Char	No.
515.625	{500, 531.25}	a	0	s	18	A	36	S	54
578.125	{562.5, 593.75}	b	1	t	19	B	37	T	55
640.625	{625, 656.25}	c	2	u	20	C	38	U	56
703.125	{687.5, 718.75}	d	3	v	21	D	39	V	57
765.625	{750, 781.25}	e	4	w	22	E	40	W	58
828.125	{812.5, 843.75}	f	5	x	23	F	41	X	59
890.625	{875, 906.25}	g	6	y	24	G	42	Y	60
953.125	{937.5, 968.75}	h	7	z	25	H	43	Z	61
1015.625	{1000, 1031.25}	i	8	0	26	I	44	!	62
1078.125	{1062.5, 1093.75}	j	9	1	27	J	45	#	63
1140.625	{1125, 1156.25}	k	10	2	28	K	46	$	64
1203.125	{1187.5, 1218.75}	l	11	3	29	L	47	%	65
1265.625	{1250, 1281.25}	m	12	4	30	M	48	=	66
1328.125	{1312.5, 1343.75}	n	13	5	31	N	49	(	67
1390.625	{1375, 1406.25}	o	14	6	32	O	50	)	68
1453.125	{1437.5, 1468.75}	p	15	7	33	P	51	+	69
1515.625	{1500, 1531.25}	q	16	8	34	Q	52		
1578.125	{1562.5, 1593.75}	r	17	9	35	R	53		
1640.625	{1625, 1656.25}	.	70	_	71	-	72		
1703.125	{1687.5, 1718.75}	Use Symbol 2							
1765.625	{1750, 1781.25}	Use Symbol 3							
1828.125	{1812.5, 1843.75}	Use Symbol 4							
1890.625	{1875, 1906.25}	Start / Stop / Separator							

Table 5.1: The mapping from frequency of the tones to symbols: 73 symbols are encoded with 24 frequencies. Therefore, some symbols are represented by two tones.

might have 10 characters. For typical application package names, the number of characters might range from 16 to 64 characters. The tone sequence for an overall message length of 32, 64 and 128 characters lasts from 4.1 to 7.68, 8.19 to 15.87 and 16.38 to 32.26 seconds. However, the length of the application package name could be shortened (and kept constant) by using a URL shortening service.

This discovery approach has high hardware requirements, because recording audio and recognizing the tones (i.e., performing a FFT) simultaneously is computational expensive. Therefore, the recognition of the tones lags several seconds behind the played tones on some devices. The Android phone "Google Nexus One" equipped with a 1 GHz CPU and 512 MB RAM running Android 2.3.4 is able to perform these tasks in close to realtime, whereas the recognition on the same phone running Android 2.2.2 lags a few seconds behind the played tones. Due to the hardware (different microphones, loudspeakers, etc.) and the alignment of the devices to each other, recognition errors might occur. These recognition errors are almost all detected with the help of the implemented checksum.

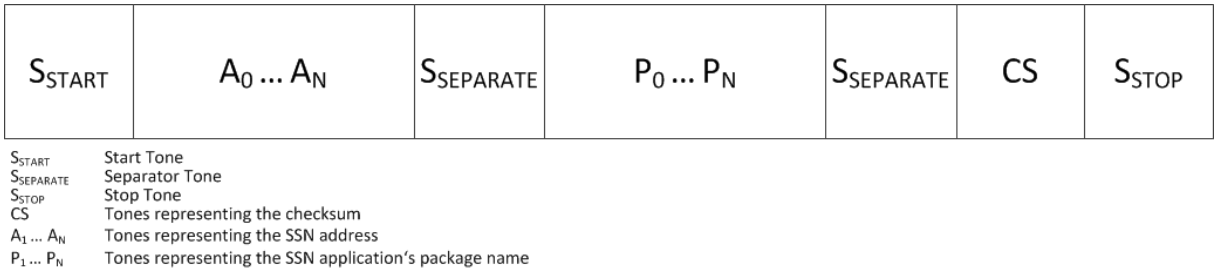


Figure 5.9: Message format for SSN discovery via tones: The transmitted message consists of a start and stop tone, tones representing the SSN address and the SSN application's package name, the checksum and separator tones for separating each message part of each other.

5.4 Implementation Issues

5.4.1 Time Leveling

Whereas in standard XMPP messages a timestamp is not included, in other messages (such as messages from delayed delivery [XSF09c], i.e., messages to offline recipients) the date and time is included. This date and time is set by the XMPP server. However, the date and time of the XMPP server is usually not equal to the date and time of the users' devices. This is due to different time zones and the fact that all clocks have not exactly the same time. Messages that are delivered immediately and those that are delivered with a delay have not the same time base. Therefore, the chronological sequence and the time span between messages might be wrong. The time zones can be easily corrected, because the time zone in XMPP is the Coordinated Universal Time (UTC) or the offset to the UTC is given [XSF03]. To achieve the correct time and date, the device's clock must be synchronized with the XMPP server's clock after connecting to the XMPP server. The Entity Time extension (XEP-0202) [XSF09a] is used for this purpose. It enables a user to retrieve the current server's time. The user's device is able to calculate the offset between the two times and adjust each received date and time. Due to the message's transmission time and the processing time there is a small error in the offset.

5.4.2 Application's Responsiveness

As discussed in Section 3.2.7 the application's main thread must not be blocked. If there is a connection timeout, the caller thread of a method in Asmack that sends data to the XMPP server will be blocked for a few seconds. The caller thread is a thread of the SSN service. However, calls to a remote service in Android are synchronous and therefore the SSN application's caller thread is blocked. Therefore, each call to Asmack that triggers sending data over the TCP connection to the XMPP server is executed in another thread. In our prototype implementation we used an own thread for each call to Asmack that triggers sending data to the XMPP server. To reduce the overhead and increase performance, a thread pool is used. The result of each method and exceptions (because they are not supported across processes) are delivered to the caller of the method with the help of a callback object due to the asynchronous method calls.

5.4.3 Connection Management

Usually the SSN applications are interested in the current online status of the connection to the server for displaying messages, disabling UI elements, changing icons, etc. A listener via IPC can be used for that, but requires frequent adding and removing of callbacks. We decided to use a broadcast intent to notify the SSN applications of the online status. The SSN applications register a broadcast receiver to listen for the online status. The activities of the SSN applications typically register a listener at the broadcast receiver to get notified of the online status. There is no simple and secure way to ensure that only SSN applications receive the online status [Bur09]. However, each SSN application must own the "use SSN" permission to bind to the SSN service and use the SSN framework. The SSN framework identifies the SSN applications with the help of this permission. The SSN applications that do not own the "use SSN" permission are not allowed to receive the online status broadcast intents.

If there is a connection loss, the SSN framework tries to reconnect when there is a connection to the internet. According to the policy in Smack, the first minute the SSN framework tries to reconnect several times, the next five minutes it tries to reconnect a few times and after this time span it tries to reconnect rarely. Each mentioned number of reconnection attempts can be specified by the users in the settings.

Some operations in SSNs are not atomic. For instance, for creating and joining an SSN several XMPP stanzas that depend on each other are sent. XMPP or stream errors might occur while sending a sequence of XMPP stanzas due to for example connection losses. Unfortunately, there is currently no public XMPP extension for this problem. Therefore, we store such failed requests in a database and send them after the next successful connection to the XMPP server. The "no response" exception is thrown in Asmack when the client does not receive an answer for a sent message within a predefined time. This exception is problematic, because the client does not know if the message has been processed successfully or not. Therefore, (nearly) all methods can be executed several times with the same parameters without changing the state and without receiving errors.

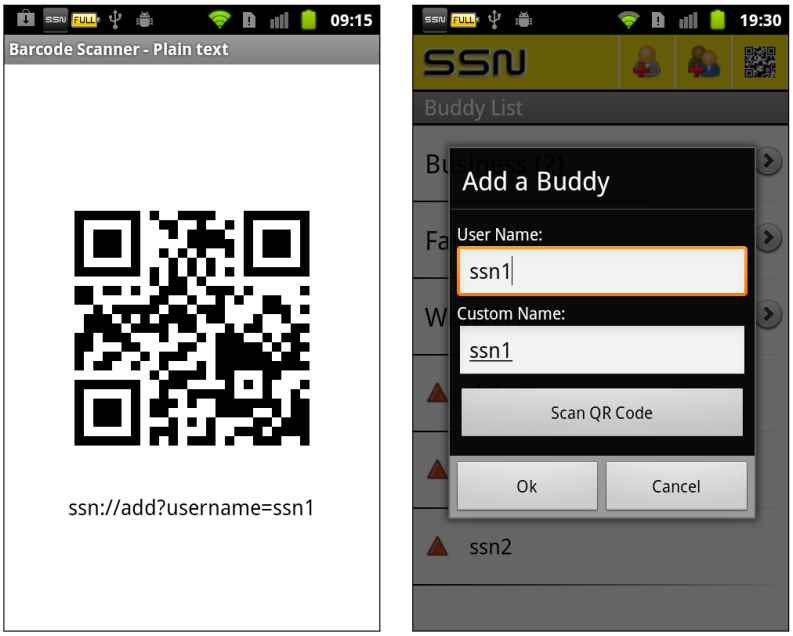
5.4.4 Sharing of the User's Identity

Users become acquainted in SSNs and are able to add each other to the buddy list. However, a user often wants to add her/his friends, colleagues or users in her/his proximity to the buddy list. For this purpose the user is able to enter the user name in a user interface, but this is cumbersome. However, the mentioned approaches for sharing and discovering SSNs can be used for this purpose. We implemented only the sharing of the user names by QR codes. The data that is displayed as QR code is an URI of the following format:
`ssn://add?username=<user name of the user>`.

The QR code for sharing the user's identity is shown in Figure 5.10 (a) and the user interface to add a buddy is shown in Figure 5.10 (b).

5.4.5 Automatic Deletion of SSNs

The SSNs have to be automatically deleted on the XMPP server when the TTL has expired. We used a cron job for deleting all expired PubSub nodes, which is implemented in ejabberd in the module `mod_cron`. The cron job is executed every hour. We think that this is sufficient, because the SSN framework also checks if the TTL of an SSN has expired.



(a) (b)

Figure 5.10: The activity for sharing the user's identity (a) and the alert dialog for adding a buddy (b): (a) shows the activity of the Barcode Scanner from the ZXing Team [Zxi11] that displays the QR Code to share the user's identity. If the user clicks on the "Scan QR Code" button in (b), the Barcode Scanner of the ZXing Team will be launched.

5.4.6 Security

There are several security problems that occurred during the implementation of the SSN framework. In this section the main problems are discussed by raising security questions.

How do we guarantee that only the SSN administration application is able to use administration methods such as deleting buddies, adding installed SSN applications and retrieving the full SSN list? Android offers permissions as described in Section 3.2.5. A signature-level permission is used to protect administration methods from being called from SSN applications. All applications of a developer are signed with the same certificate. Obviously, this is a backdoor for the developer of the SSN framework to call administration methods from her/his SSN applications, but the SSN framework developer does not want to harm her/his own framework. Another pitfall occurs when securing AIDL methods. The permissions cannot be declared in the manifest and it is not possible to protect an entire AIDL interface. The latter would be feasible by checking the permission in the remote service's `onBind` method. However, the binder's Linux user ID and process ID that are needed to check a permission cannot be retrieved in the onBind method. Instead of that, the remote service's user ID and process ID is returned. This leads to checking the remote service itself of owning the signature-level permission, which is in our case always true. Thus, the permission must be checked at the beginning of each administration method.

How does the SSN service guarantee that the SSN application is really the expected application? The SSN framework acts as a delegate for the SSN applications. Therefore, it is necessary that the SSN framework is able to identify the caller of the method. It is not possible to retrieve the caller's package name from a method of the AIDL interface in a remote service, but it is possible to retrieve the caller's Linux user ID and process ID. Both IDs are provided by the kernel [Bur09]. If the caller declares its package name as a method parameter, the SSN service will be able to validate the given package name.

That is, the service queries the Android package manager for the Linux user ID and compares it with the caller's Linux user ID. The Linux user ID is assigned by Android during the installation. Applications of the same developer (i.e., signed with the same certificate) will run with the same Linux user ID if it is specified in the Android manifest file. However, this is again not a problem, because a developer will not harm herself/himself. A related problem is that an evil SSN application is able to have the same package name than an SSN application. Two applications with the same package name cannot be installed at the same time. All applications should be installed from the Android Market, because it ensures that each package name is unique in the Android Market. Therefore, all requests to install necessary SSN applications are linked to the Android Market. A similar problem occurs when launching activities from the SSN applications. The caller's package name can only be retrieved when starting activities synchronously (i.e., the sending activity waits on a result). However, activities that require the package name of an SSN application such as sharing an SSN or displaying the SSN details are private and can only be launched by calling a method in an IPC interface. The method checks the package name and launches the activity in a new task.

How do the SSN applications guarantee that the intent for joining and entering an SSN is really sent from the SSN administration application? If a user wants to join an SSN or enter an SSN from the SSN administration application, the SSN administration application sends an intent to the responsible SSN application. The SSN application must ensure that the sender of the intent is the SSN administration application. The sender of an intent can only be determined at the receiver if an activity is started synchronously (i.e., the sending activity waits on a result). However, permissions can be used for this purpose. The activity or activities used for joining and entering SSNs are protected by the signature-level permission that is used to protect the service's administration methods from being called from SSN applications. Therefore, only the SSN administration application (and applications that are signed with the same certificate as the SSN administration application) is able to launch the activities of the SSN applications for joining and entering an SSN. The SSN applications are not able to launch these activities, because they are not able to own the signature-level permission of the SSN administration application.

5.5 XMPP Server

5.5.1 Evaluation

There are plenty of XMPP server implementations. Most of the XMPP extensions require server support. Due to the huge number of extensions, not all server implementations support all extensions. Thus, a good choice for a server has to support the required extensions. Furthermore, it should have a good documentation, and an active community, should be open source software, free to use, and used in many projects. Only ejabberd and OpenFire fulfill these requirements at least partially.

Ejabberd is written in Erlang. Erlang [Arm10] was developed by Ericsson 1986 for telecommunication. It offers useful features such as concurrency, distribution, robustness, changing code during runtime and several libraries to implement distributed, fault-tolerant, real-time applications. Ejabberd is used for many public XMPP services (servers).

OpenFire is the successor of Wildfire, which was a commercial server implementation. It is written in Java. OpenFire is developed by Jive Software, which develops also the client library Smack.

Table 5.2 lists the required extensions for SSNs and compares which server implementation supports which extension. The comparison is based on the respective documentation [Eja11, Jiv11]. Table 5.3

compares the supported features of the PubSub extension. This comparison is obtained by testing.

Extension	ejabberd 2.1.6	OpenFire 3.6.4
XEP-0004 Data Forms	Yes	Yes
XEP-0016 Privacy Lists	Yes	Yes
XEP-0030 Service Discovery	Yes	Yes
XEP-0059 Result Set Management (RSM)	Yes	Yes, but not in PubSub
XEP-0060 Publish-Subscribe	Yes	Yes
XEP-0077 In-band Registration	Yes	Yes
XEP-0082 XMPP Date and Time Profiles	Yes	Yes
XEP-0115 Entity Capabilities	Yes	Yes
XEP-0158 CAPTCHA Forms	Yes	No
XEP-0163 Personal Eventing Protocol	Yes	Yes
XEP-0202 Entity Time	Yes	Yes
XEP-0203 Delayed Delivery	Yes	Yes

Table 5.2: Comparison of the supported extensions of ejabberd 2.1.6 and OpenFire 3.6.4 [Eja11, Jiv11]

Extension	ejabberd 2.1.6	OpenFire 3.6.4
Support for Result Set Management XEP-0059	Yes (only ODBC version)	No
Associating Events and Payloads with the Generating Entity	No	No
Node Meta-data	No	Yes
pubsub#send_last_published_item or pubsub#send_item_subscribe	Yes	Yes
Offline Storage of PubSub event notifications	Yes	No
Config / Meta-data pubsub#language and pubsub#description	No	Yes

Table 5.3: Evaluation of the supported PubSub (XEP-0060) features of ejabberd 2.1.6 and OpenFire 3.6.4

5.5.2 Conclusions

Even though OpenFire is written in the well-known Java programming language and therefore is for us easily extensible, we decided to use ejabberd. Reasons for this decision are, that ejabberd supports all required XMPP extensions, is faster, more scalable and is used for many public XMPP services. The implementation of the missing extensions and features in OpenFire would be rather time-consuming. Drawbacks using ejabberd are that some required PubSub features are not supported. Therefore, we had to extend ejabberd. The implementation of the required features for ejabberd is quite involved, and requires (less common) knowledge about the Erlang language.

5.6 XMPP on Android

To realize the communication over XMPP in SSN, a XMPP client library is required. Either a library for Android or for JavaScript might be considered. If a JavaScript library is used, the SSN framework and the example applications must be implemented by web technologies such as JavaScript, HTML and CSS.

However, accessing the device's hardware such as sensors and accessing other Android features from this software layer is difficult. Hence, we decided to use a Java library for the Android implementation.

The open-source library Asmack is an adapted version of the Smack library for Android. It is based on Smack 3.1.0.

Smack is implemented in Java. It provides all functionality for using the XMPP core protocols and some extensions. The extensions are realized by the provider architecture. Each extension is able to define its own packets. A packet is a stanza or a child of the XML root tag. The corresponding providers are responsible for parsing these packets. The big advantage of this approach is extensibility. Each required provider has to be registered by the application to improve the parsing performance. Smack provides a lot more functionality such as packet listeners and interceptors for incoming and outgoing packets including filtering.

Asmack supports several XMPP extensions such as Service Discovery, File Transfer and Data Forms. The PubSub extension is also supported, but not all PubSub features of the latest PubSub version are implemented. PEP is supported, but there are a lot of problems with the implementation. Therefore, and due to some other bugs, we decided to patch and extend Asmack. All modifications are documented in Appendix D.

Chapter 6

Example Applications

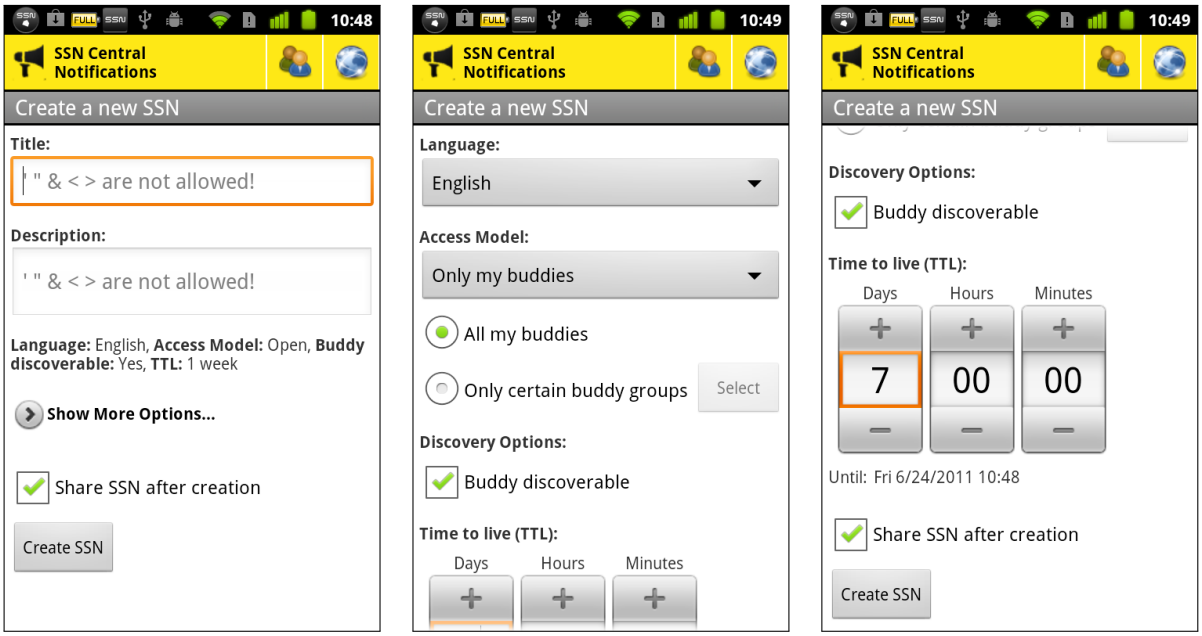
6.1 Overview

The example applications demonstrate the benefits and features of the SSN framework. Furthermore, they test and evaluate the SSN framework. The following three applications are implemented for testing purposes of the different communication models.

6.2 Central Notifications

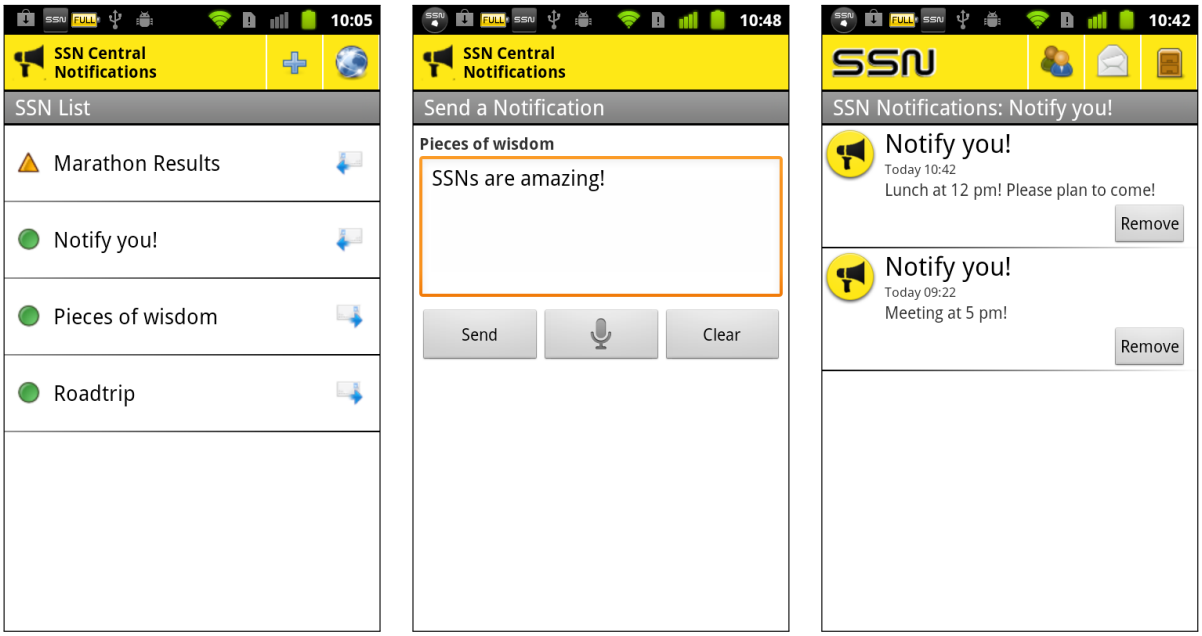
In the Central Notifications application one user, the SSN owner, is able to send text messages to an interested audience. This application is very useful in a lot of scenarios. For instance, the head of a tour group can use this application to notify all participants of events such as the time for lunch, the time for the next meeting, the departure time for the next sightseeing tour and the time and the location for the upcoming party. A boss is able to notify her/his employees of the time for lunch, time for the next meeting or current events. A chairman of a club can notify the club's members of upcoming tours, trips, meetings or LAN parties. A user is able to use this application for micro-blogging to her/his friends. A family member can notify the rest of the family of the food someone has to buy for the lunch, the arrival at home or current events.

The application offers user interfaces to create (see Figure 6.1) and manage (see Figure 6.2 (a) and Figure 6.3 (a)) central notification SSNs. Furthermore, a user interface for sending new notifications is implemented as shown in Figure 6.2 (b). The notifications are received and displayed by the SSN notification system as shown in Figure 6.2 (c). To control the access to an SSN. all supported access models of the SSN framework are available in the Central Notifications application. Obviously, this application uses the 1 to N communication model and one node per SSN. The SSNs can be discovered by all available discovery approaches as shown in Figure 6.3 (b). The XML schema of the PubSub items is shown in Appendix C.



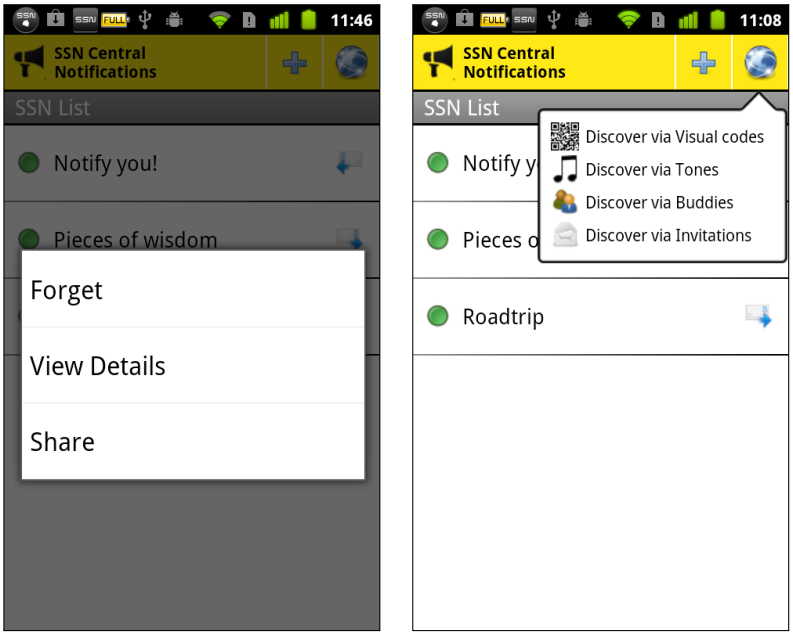
(a) (b) (c)

Figure 6.1: The activity for creating new Central Notifications SSNs: (a) shows the initial activity for creating new SSNs. If the user clicks on the "Show More Options" button, all available options will be displayed as shown in (b) and (c).



(a) (b) (c)

Figure 6.2: The activities for displaying all joined Central Notifications SSNs (a), sending a notification in an SSN (b) and displaying the notifications of an SSN (c). (a) is the first activity that is displayed when opening the Central Notifications application. The orange triangle indicates that at least three-quarter of the TTL is over. The icon next to the SSN indicates if the user is only able to send or receive notifications. If the user is the creator of the SSN and clicks on an SSN item, the activity to create a new notification (b) will be displayed. Otherwise the notifications activity of the SSN administration application will be launched and only the notifications of the clicked SSN are displayed. (b) and (c) are different SSNs!



(a) (b)

Figure 6.3: The activities for managing joined SSNs (a) and discovering new Central Notifications SSNs (b): If the user clicks on the action button on the right, all available discover options will be displayed as quick actions as shown in (a). (b) shows the outcome if a user performs a long touch on an SSN item. The user is able to forget (leave, unsubscribe) the SSN, display the SSN details or share the SSN.

6.3 YouTube Shoutwall

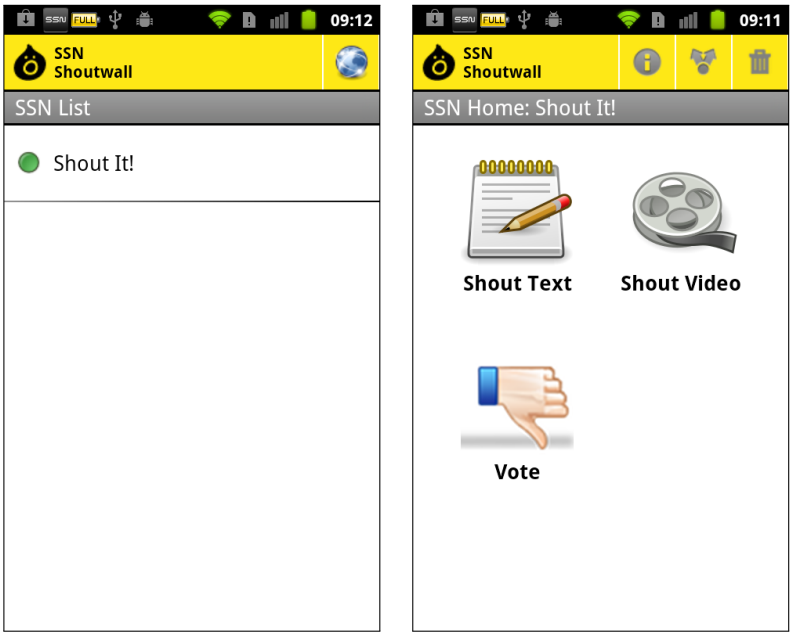
The YouTube Shoutwall enables users of an SSN to post text messages and IDs of YouTube videos on a shoutwall. If a video is posted while another video is played, the posted video will be queued. After the end of the currently played video, the video on top of the queue is played. To improve the readability of the shoutwall, new text messages are only displayed in a minimum interval of 10 seconds. Furthermore, the SSN users are able to vote for or against the current video. If the votes against the video exceed the votes for the video by a predefined voting threshold, the current video will be stopped and the next queued video will be played. Each user is only able to vote once per video (i.e., the first vote counts). The shoutwall can for instance be projected on large screens or shown on TV sets. It can be used for entertainment in bars, on parties and the entrance hall of a university, school or hotel. The YouTube Shoutwall could also be integrated on a website.

Actually, the YouTube Shoutwall consists of two applications. One application is a client bot implemented in Java and the other one is an Android application. The user of the client bot is the owner of one specific SSN and its corresponding shoutwall. It is used for managing the voting and displaying the posted videos, text messages and the SSN's QR-code as shown in Figure 6.4. The client bot is implemented in Java, JavaScript and HTML. A web browser is integrated in Java's user interface library Swing with the help of the DJ Project's Native Swing library [Dec11]. We use the YouTube's JavaScript library [Goo11b] for playing the videos and fetching the video's metadata.

The Android application implements activities for displaying and managing all joined SSNs as shown in Figure 6.5. It enables the users to post text messages (see Figure 6.6 (a)), search and post YouTube videos (see Figure 6.6 (b)) and vote for or against the current played video (see Figure 6.6 (c)). A user is also able to use the Android YouTube application for searching and picking videos by sharing it with the SSN Shoutwall application. The SSN Shoutwall application announces an activity with an intent filter (ACTION_SEND, mimeType: text/plain). If the user shares a video with the SSN application, the YouTube application will call the announced activity and will pass the YouTube URL to it.

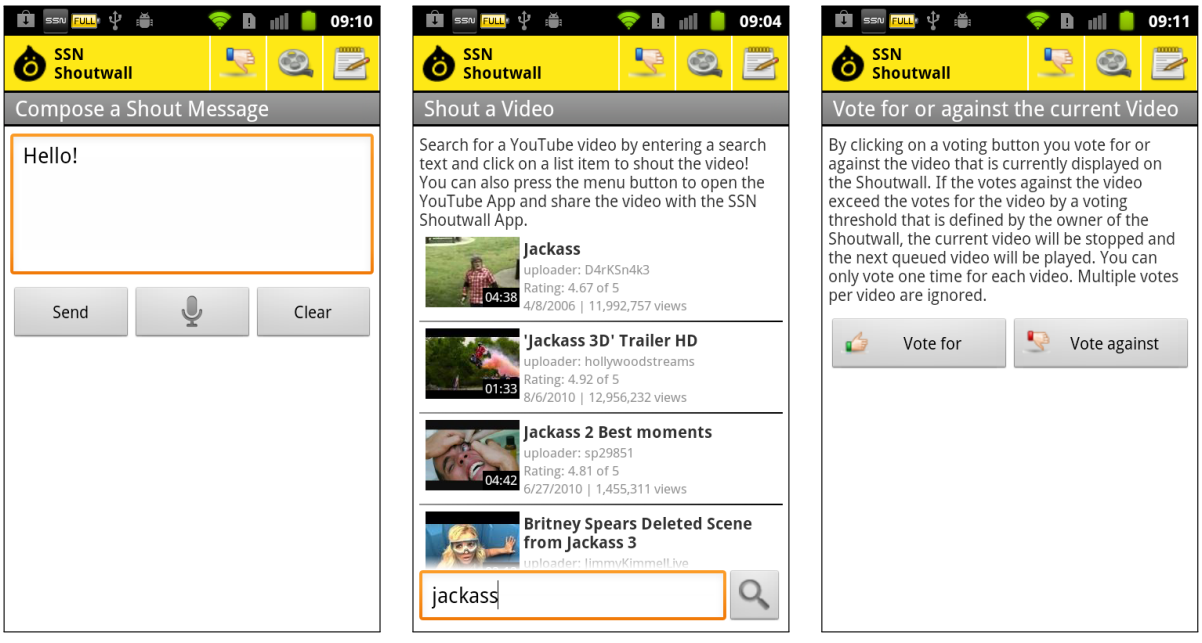
All supported access models of the SSN framework are available during the creation of new YouTube Shoutwall SSNs. The YouTube Shoutwall uses the N to 1 communication model and requires one node per SSN. The privacy model is semi-anonymous. The users discover YouTube Shoutwall SSNs by the displayed or suspended QR code. The XML schema of the PubSub items is shown in Appendix C.

Figure 6.4: YouTube Shoutwall bot: It displays all received text messages and plays the received YouTube videos. The QR Code is displayed on the left and right side to join the SSN. The yellow rectangle shows the votes for this video. Yellow indicates that the number of votes for and against the video are equal. If the voting threshold for/against the video is reached, the rectangle's color will be green/red. The rectangle's color is interpolated between green and red for values in between.



(a) (b)

Figure 6.5: The activity for displaying all joined YouTube Shoutwall SSNs (a) and the SSN home activity (b): If the user clicks on an SSN item in (a), the SSN home activity (dashboard) (b) is launched. The SSN home activity shows all available functionality of an SSN and action buttons for displaying the SSN details, to share and forget (leave, unsubscribe) the SSN.



(a) (b) (c)

Figure 6.6: The activities for shouting text messages (a), YouTube videos (b) and voting for or against the video (c) on the YouTube Shoutwall. If the user clicks on a video in (b), the video will be shared after acknowledging a confirmation dialog.

6.4 Event Scheduling

The Event Scheduling application enables users to find a date for an event collaboratively. An event can be any appointment where people meet such as a business meeting, the date of a journey, the date of the lunch with friends or the date of painting the fence. The features of this application are similar to the features of the widely-used web service Doodle [Doo11]. However, the event scheduling application has some additional benefits, because it

- provides a chat
- enables all users to suggest new dates
- provides the strengths of SSNs (e.g., access models, privacy models, smart discovery approaches, realtime communication)

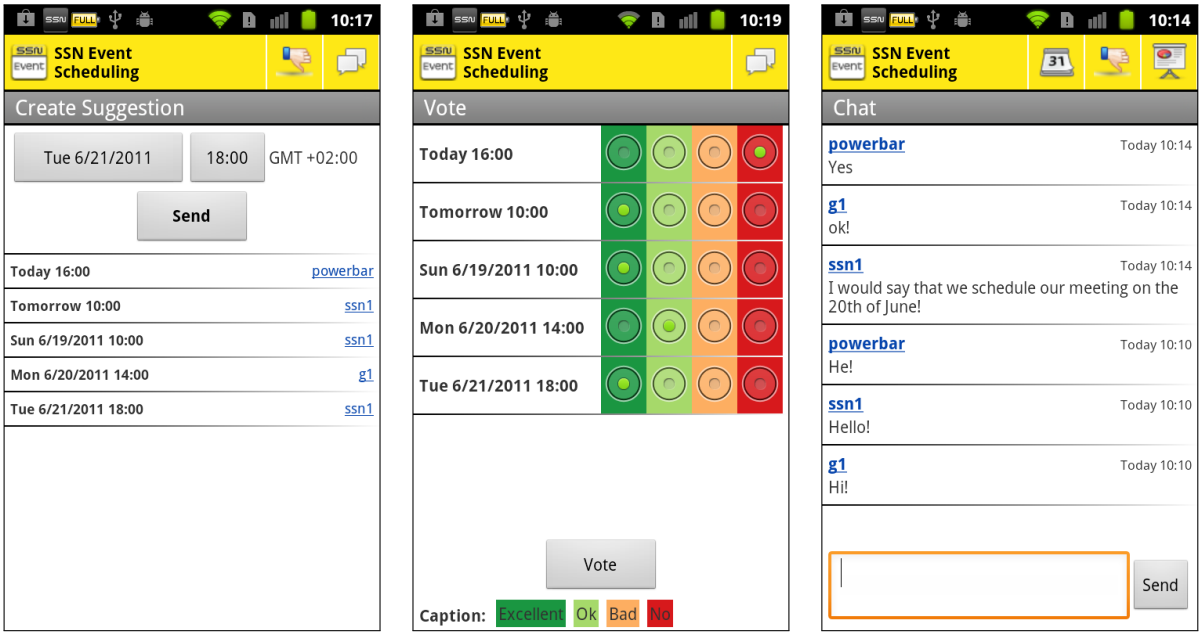
Each user is able to create suggestions as shown in Figure 6.7 (a). They can vote for the suggested dates at any time and the voter is able to modify her/his vote whenever (s)he wants. There are five choices for each suggested date as shown in Figure 6.7 (b): Excellent date for the user, the date is ok for the user, bad date for the user, the user does not have time or the user has not voted for the date. The chat should be used to discuss the suggested dates, to exchange further information and to fix the date as shown in Figure 6.7 (c). Furthermore, the Event Scheduling application visualizes the result by bar and pie charts as shown in Figure 6.8. The members are able to retrieve the user names of the voters. The Event Scheduling application provides a suggestion for the final date. This suggestion is based on the number and kind of votes for each date. The following criteria are processed consecutively:

- The date with the fewest votes against it (i.e., labeled "no")
- The date with the fewest votes labeled "bad date for the user"
- The date with the most votes labeled "excellent"
- The date with the most votes labeled "ok"

All dates are matched against the first criterion. If only one date fulfills the criterion, this date will be suggested. Otherwise, the dates that fulfill this criteria are tested against the next criteria. If there is more than one suggestion after the last criterion, all dates that fulfill the last criterion are suggested.

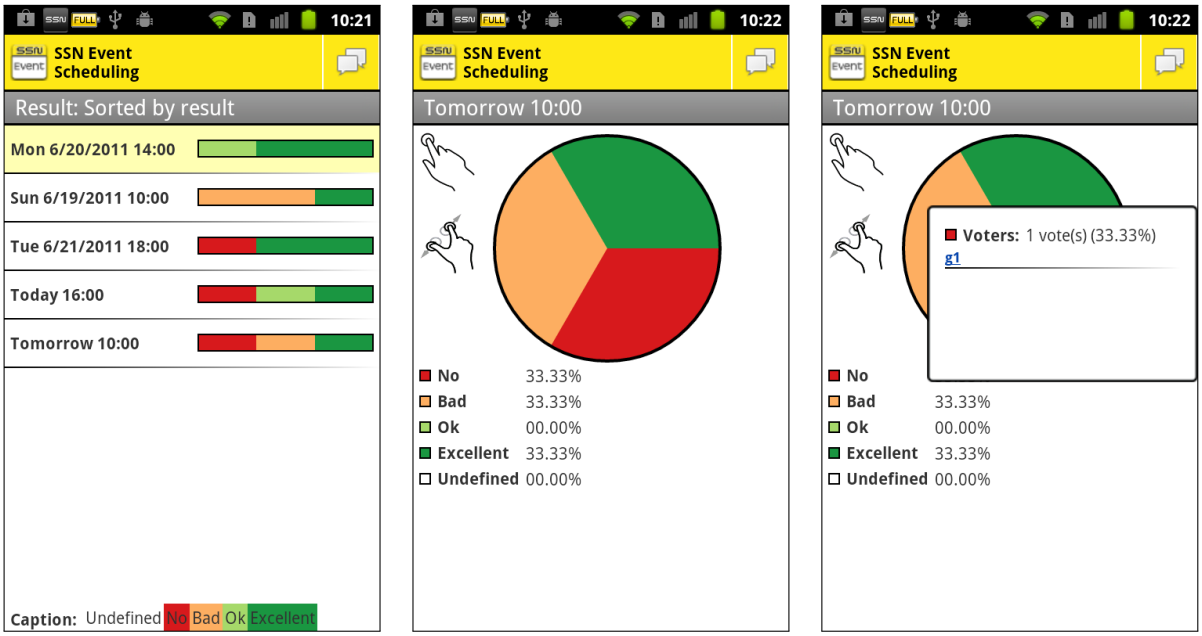
The Event Scheduling application implements activities for displaying and managing all joined SSNs as shown in Figure 6.9. All supported access models of the SSN framework are available during the creation of new Event Scheduling SSNs (the activity for creating new SSNs looks similar to Figure 6.1). Furthermore, the SSN creator is able to specify the title, description, language, TTL and if the SSN is buddy discoverable or not. The Event Scheduling application uses the N to N communication model and requires three nodes per SSN. Each of the features (chat, suggestions and voting) requires one node. The messages of the chat node are delivered to offline users, but not stored on the node. The suggestions and the votes are stored on the server, because otherwise new members would not have access to all suggestions and votes. The messages of both nodes are delivered to offline users, because PubSub does not offer a feature to retrieve only new items (i.e., items created or updated in a certain time span) from a PubSub node. The votes and suggestions can be stored in principle on one node, but for scalability and encapsulation reasons we decided to use two nodes. In ejabberd a maximal number of stored items per node must be specified or a default value of 20 is retained. The maximal number for this option is

limited by the ejabberd configuration file. This fact limits the scalability of the Event Scheduling SSNs, because there is a maximal number of SSN members that are able to vote. Therefore, the votes of each user are stored in one item. The item ID of a user's vote on the voting node is the user's user name to provide a simple way to modify the user's vote. Anonymous Event Scheduling SSNs are not allowed to ensure that each SSN member votes only once. The chat messages, votes and suggestions are cached on each user device in a database for increasing the performance and providing a history for the chat. The users discover Event Scheduling SSNs by the displayed or hung up QR code, by invitations and in cliques with the help of the buddy based discovery approach. The XML schema of the PubSub items is shown in Appendix C.



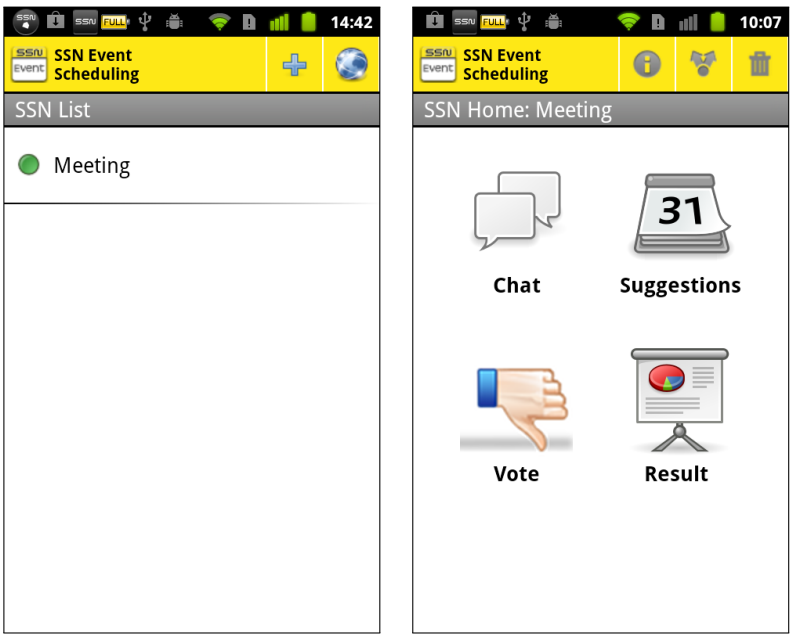
(a) (b) (c)

Figure 6.7: The activities for creating suggestions (a), voting (b) and the chat (c) of the Event Scheduling application.



(a) (b) (c)

Figure 6.8: The activities for visualizing the result of the Event Scheduling application: (a) shows the results of all suggestions. The date "Mon 6/20/2011 14:00" is suggested by the application. (b) shows the detailed result of one suggestion with the help of a pie chart. If the user clicks on a sector of the pie chart, the voters of the corresponding vote option are shown as seen in (c).



(a) (b)

Figure 6.9: The activity for displaying all joined Event Scheduling SSNs (a) and the SSN home activity (b): If the user clicks on an SSN item in (a), the SSN home activity (dashboard) (b) is launched. The SSN home activity shows all available functionality of an SSN and action buttons for displaying the SSN details, to share and forget (leave, unsubscribe) the SSN.

Chapter 7

Evaluation

7.1 Usability

7.1.1 Overview of the System Usability Scale

The System Usability Scale (SUS) was introduced by Brooke [Bro96]. It is "a reliable, low-cost usability scale that can be used for global assessments of systems usability." [Bro96]. The result of SUS is a single number that represents the system's overall usability. SUS is performed with a small group of people. Tullis and Stetson [TS04] analyzed different usability scales including SUS and state that a group of at least 12-14 participants is required to achieve reliable results. First of all, the participants have to use the system. Before any discussions, the participants have to answer a ten-items questionnaire by specifying the degree of agreement or disagreement to given statements from strongly disagree (scale position 1) to strongly agree (scale position 5). This type of scale is called a Likert scale. Each question should be answered immediately without thinking for a long time. If a user is not able to answer a question, (s)he should mark scale position 3. The resulting SUS score is determined by summing up the points of each question and multiplying it by 2.5 to achieve a final SUS score in the range of 0 to 100. The points for each question are the given scale position minus 1 in cases where strong agreements means good usability (questions 1, 3, 5, 7, 9) and 5 minus the scale position otherwise (questions 2, 4, 6, 8, 10). A high SUS score means good usability. Figure 7.1 shows the SUS questionnaire.

7.1.2 Results

We performed two SUS tests with two different groups. The first test was scheduled for a beta version of our prototype implementation. The implementation contained some known bugs and some missing buttons. The test was executed with 6 computer graphics male students. The students attended a course where they had to implement a virtual reality game on Android.

The second test was performed with 9 persons (3 females and 6 males, 8 students and 1 apprentice, from 21 years until 27 years old) with the completed prototype implementation. 5 of them own an Android phone or attended courses where they had to develop Android applications. The other students had experiences on other mobile platforms such as iPhone OS and Windows Mobile.

None of the participants has used SSNs before. Therefore, we give a short introduction talk to describe the purpose and aim of the SSN framework and the implemented example applications. We defined a

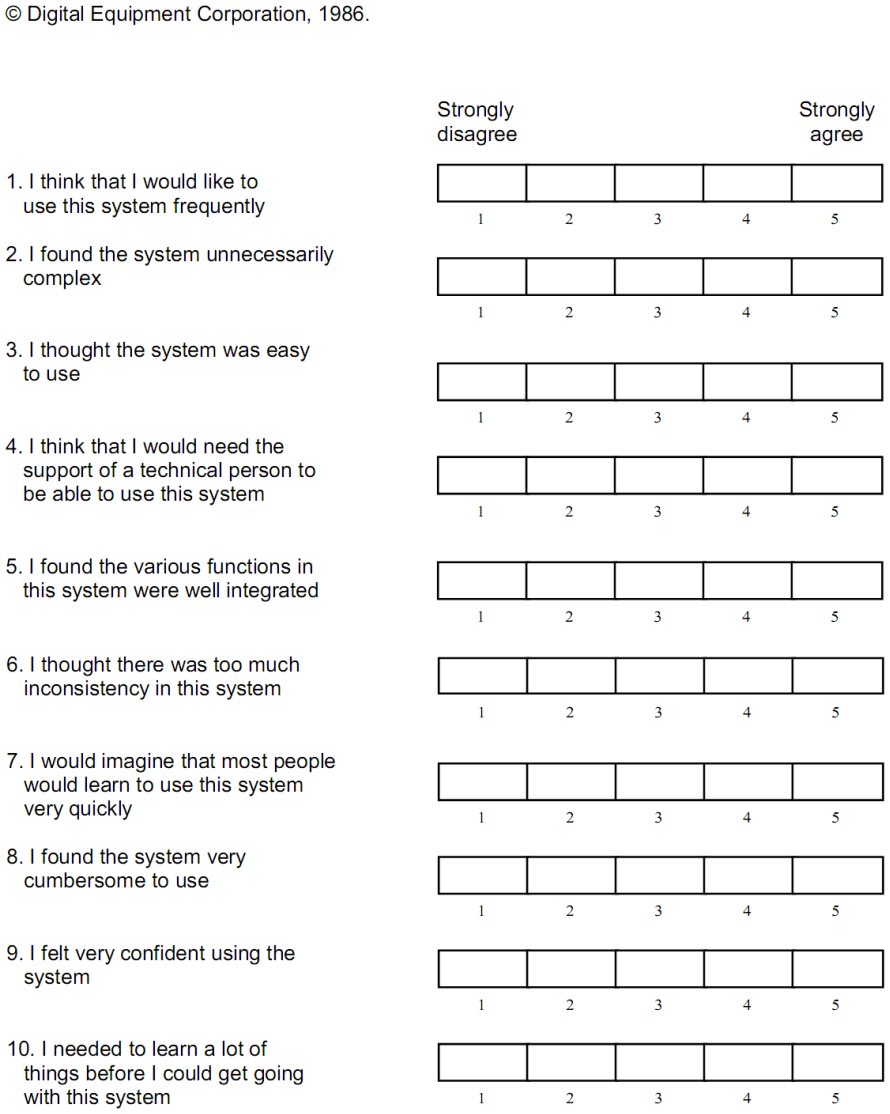


Figure 7.1: SUS questionnaire consists of 10 statements (questions). Each user has to specify her/his degree of agreement or disagreement to the statements. At statements with an odd number "strongly agree" means good usability, whereas at statements with an even number "strongly disagree" means good usablility. [Bro96]

scenario for the tests and created an SSN for Event Scheduling and for the YouTube Shoutwall. Except the installation and setup procedure we tried to integrate the most frequently used tasks in the scenario as suggested by Bangor et al. [BKM08]. After the installation and setup procedure, the participants joined the SSNs with the help of QR codes. Each student suggested one date, voted for the suggestions, used the chat and viewed the result. On the YouTube Shoutwall SSN the participants posted videos and chat messages. The participants did not use all implemented features, because that would have been too time-consuming. Furthermore, the participants did not use the Central Notifications application.

The result of the final SUS test is 80.83 of 100 points, whereas the SUS test of the beta version scores only 64.58 points. Bangor et al. [BKM08, BKM09] performed 206 SUS tests with 2,324 participants in

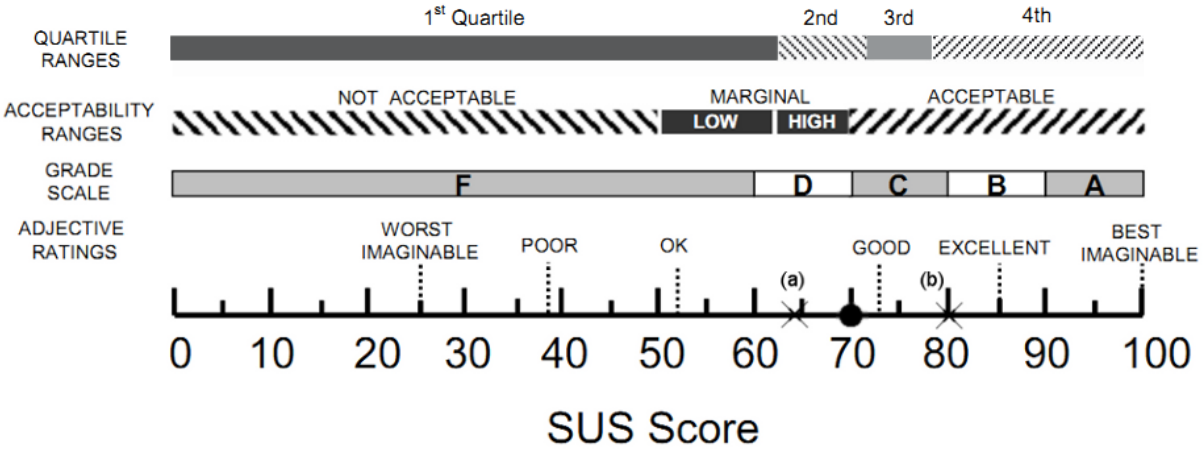


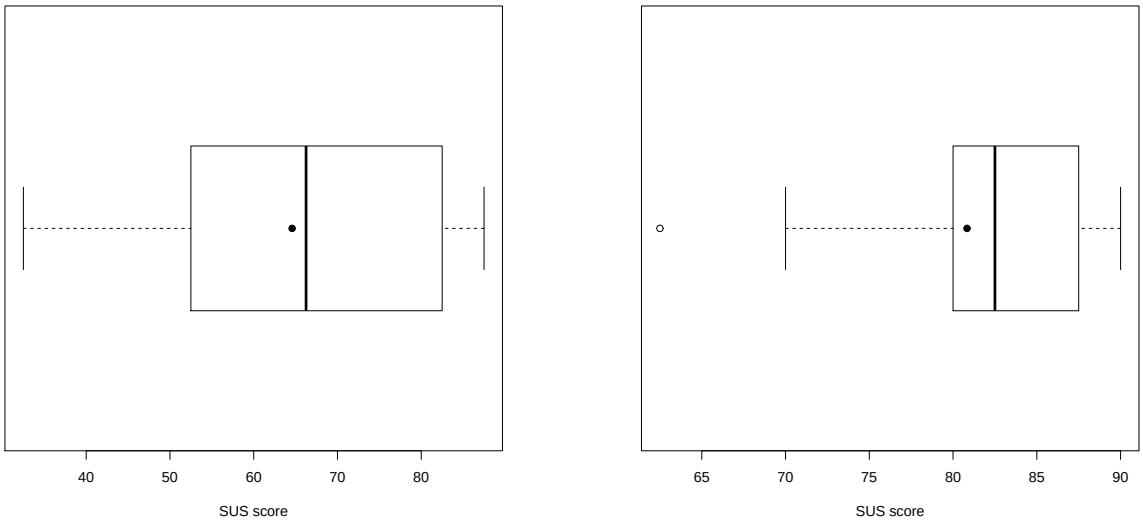
Figure 7.2: The quartile ranges, acceptability ranges, grade scale and adjective ratings based on the analysis of Bangor at el.: The result of the SUS test of the beta version of SSNs is marked with (a) and the result of the final version of SSNs is marked with (b). The filled circle marks the mean of all tests performed by Bangor at el. [BKM08, BKM09]

	SUS tests of SSNs		Analysis of Bangor et al.	
	Beta version	**Final version**	**Per SUS test**	**Overall**
Mean SUS score	64.58	80.83	69.69	70.14
Standard deviation	19.06	8.74	11.87	21.71
Minimum SUS score	32.50	62.50	30.00	0.00
Maximum SUS score	87.50	90.00	93.93	100.00
Median SUS score	66.25	82.50	70.91	75.00
Adjective rating	Ok	Good	Ok	Ok
Acceptable range	Marginal high	Acceptable	Acceptable	Acceptable
Quartile range	2nd	4th	2nd	2nd
Grade scale	D	B	D	C
Number of participants	6	9	11.28 (averaged)	2,324
Approximate accuracy	35%	75%	90%	100%

Table 7.1: The results of the two performed SUS tests compared with the analysis of Bangor et al. [BKM08, BKM09]: The maximal achievable points of each SUS test is 100. The adjective rating, acceptable range, quartile range and grade scale are introduced by Bangor et al. The approximate accuracy was analyzed by Tullis and Stetson [TS04]. The SUS score of the final test means that the prototype implementation of SSNs belongs to the better products [BKM08].

nearly 10 years. The tests were performed with software running for example on phones, modems, PCs, web pages and web applications. The mean SUS score of these tests is approximately 70 points. The final SUS result of SSNs lies above the 4th quartile (i.e., in the 4th quartile range). Bangor et al. state that "products which are at least passable have SUS scores above 70, with better products scoring in the high 70s to upper 80s. Truly superior products score better than 90." They defined a grade scale (our final SUS test score: B) based on the grade scale used at school and university. Furthermore, they defined an adjective rating by performing SUS tests that include an 11th question: "Overall, I would rate the user-friendliness of this product as" with the answers worst imaginable, awful, poor, ok, good, excellent, best imaginable with 212 participants. Thus, they determine mean SUS scores for each adjective rating. The

result of the final test of SSNs lies between good and excellent. All these scales and ratings are shown in Figure 7.2. Tullis and Stetson [TS04] analyzed the required number of participants to yield an accurate result. Due to our limited resources and two short-termed cancellations we did not reach the necessary number of 12 participants to have an accuracy of nearly 100%. The results of the two SUS tests are shown in Table 7.1. Moreover, it includes a comparison with the analysis of Bangor et al. Whereas the data of the fourth column of this comparison ("Per SUS test") was obtained by examining the SUS score of each test, the data of the fifth column ("Overall") was obtained by examing all questionnaires of all tests without considering the final result of each test. Figure 7.3 shows a boxplot of the SUS score of each questionnaire and Figure 7.4 shows a boxplot for each SUS question. However, Brooke [Bro96] states that "scores of individual items are not meaningful on their own". The boxes in these figures represent the interquartile ranges. The median is shown as a vertical bold line and the mean as a filled circle. The end of the whiskers (lines) represent the maximum/minimum values or 1.5 times the interquartile range. The values that are greater/lower than the end of the whiskers are outliers, which are shown as empty circles.



(a) (b)

Figure 7.3: Box plot of the SUS score for each questionnaire: (a) shows the test of the beta version and (b) the final test. (a) has a much higher standard deviation than (b). In (b) most of the SUS scores are about or higher than 80. There is one outlier at 62.5 that decreases the mean.

We think that the worse result of the beta version is due to bugs, some missing features and buttons in the beta version (e.g., direct YouTube video search in the YouTube Shoutwall application). Therefore, we state that the questionnaires of the tests cannot be merged to yield one mean SUS score. We think that the high standard deviation of the beta version test at some questions such as questions 3 and 8 in the beta version is due to bugs that did not appear on every device and due to the small number of participants. The significant outlier at the final SUS test prohibits a better SUS score as shown in Figure 7.3. At the final SUS test there is a significant outlier at question 4 (see Figure 7.4). We think that one person made a mistake, because the participant strongly agreed with question 3. However, we did not correct this mistake.

After a short time of thinking all participants were able to use SSNs. They explored the implemented features and were able to use new features without thinking a long time. The participants were very

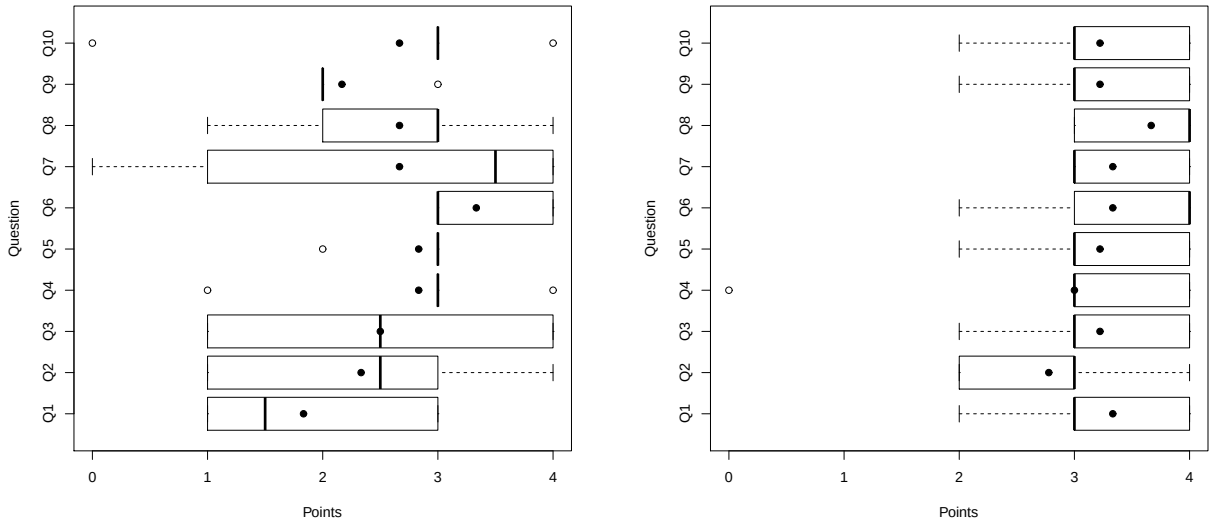


Figure 7.4: Box plot of the SUS points for each question: (a) shows the test of the beta version and (b) the final test. (a) has a much higher standard deviation at each question than (b). In (b) there is one significant outlier at question 4.

enthusiastic using the YouTube Shoutwall and had fun to share and vote for or against the posted videos.

7.2 Scalability

There is a single point of failure due to the centralized architecture of SSNs. However, the SSN architecture can be extended to achieve decentralization by using several XMPP servers (e.g., all public XMPP services). In practice this is a problem, because each server must implement all required XMPP extensions and the mentioned modifications. Furthermore, the user names have to be unique over all used servers, which is currently not the case. The SSN user name is currently the node identifier of the user's JID. Many XMPP server implementations such as the used ejabberd are able to run on a cluster of machines. The advantages of the cluster approach are distribution and fault-tolerance.

Another problem is the huge number of required PubSub and PEP nodes. The PubSub and PEP nodes are managed by the PubSub service on the XMPP server and stored in a database. In ejabberd the number of possible PubSub and PEP nodes is limited by the hardware and the database engine. We used MySQL as a database engine. The MySQL reference manual states: "Support for large databases. We use MySQL Server with databases that contain 50 million records. We also know of users who use MySQL Server with 200,000 tables and about 5,000,000,000 rows." [Ora11]. However, due to the short life span of an SSN these limits will hardly be reached in practice.

If PubSub nodes are configured to deliver events to offline users, a huge number of offline messages will be stored on the server and delivered to the offline users after the login. Whereas the storage limits have already been discussed, the delivery of offline messages to the user is limited by the bandwidth and server performance. We performed tests of delivering 10,000 offline messages, which are delivered in milliseconds to a few seconds. The short life span of SSNs reduces the number of offline messages,

because after an offline time that is greater than the maximum life span of all joined SSNs there will not be any delivery of SSN events to the user. In ejabberd the maximal number of offline messages per user can be limited in the configuration. We do not use this feature, because important messages may be lost.

Messages can be stored on a PubSub node. Whereas the query of a large number of items is not a problem due to the Result Set Management extension, the maximum number of storable items is a problem. The PubSub specification does not state if an unlimited number of items can be stored on a PubSub node. Ejabberd defines a maximum number of items.

7.3 Suitability of XMPP

The SSN approach is built on top of XMPP. XMPP is responsible for the message exchange. It solves many low-level problems such as securing messages, creating and maintaining sessions, exchanging messages in close to realtime and addressing the entities. Much functionality required for SSNs are mapped to the used PubSub extension [XSF10b], but there are some problems and limitations. Whereas the Publish-Subscribe paradigm is suitable for SSNs, the XMPP's PubSub extension has to be patched and extended to meet the requirements of SSNs.

The node owner administrates the PubSub node and is therefore for instance able to censor content, subscribe new users, remove current users, modify the access model and delete the node. This approach raises presence and security issues. There are features that require the intervention of the node owner such as modifying the affiliations and approving authentication requests. Even though SSNs are basically realtime, we cannot assume that the node owner is always online (due to connection losses, etc.). If the SSN owner is offline nobody would be responsible for processing these requests. But this is not admissible in realtime systems. If the node owner was trustable and known by each user, the owner's capabilities would not be problematic. However, in SSNs the members do not always know each other and therefore the nodes should be managed by a central authority or by all members.

Unfortunately, there are some more security issues concerning the PubSub specification. Some SSN scenarios require to store items on the PubSub node. Each publisher is able to delete items of all publishers. For SSNs each publisher should only be allowed to delete her/his own items. PubSub allows publishers to specify the ID of the stored item. The PubSub extension does not specify what the PubSub service should do if the latest published item has the same ID than an existent item. For instance, OpenFire assigns another ID to the latest published item. In contrast, ejabberd deletes the old item and stores the new item with the specified ID.

Associating events and payloads with the generating entity is only an optional feature in PubSub. That is, the PubSub service need not support this feature. The PubSub service defines which entity is able to receive the item's publisher JID, but the node owner is allowed to modify this value. The PubSub specification defines therefore only the affiliations owner and publisher. That is, at least the owner does always receive the item's publisher JID, which is a privacy issue. In case of an open publisher model all subscribers are allowed to publish without owning the publisher affiliation. Therefore, they are not allowed to receive the item's publisher JID. A solution for this problem would be to assign each new member the publisher affiliation, but only the owner is allowed to manage affiliations.

PubSub offers several access models. They are used to protect the SSN data from unauthorized access. The presence and roster access model offer great facilities for building appealing applications and are especially useful for applications that offer the ability to work in well-known groups. However, an access model that defines a global password for a node is currently missing. Even though a global password is

insecure, because the node owner cannot control the spreading of the password, it provides a basic access restriction. The password may be included in the smart discovery approaches (i.e., visual codes and tones) in SSNs. Furthermore, PubSub could also provide some sophisticated access models for instance based on the proximity of the PubSub node, Morse code based on light or vibration and acceleration of a phone. However, these access models can be realized through the "global password" access model.

PubSub offers several predefined affiliations. It would be helpful for SSNs to define new affiliations and define the privileges of the users per PubSub node. Furthermore, the affiliations can only be managed by the owner. It would be helpful for SSNs to define a default affiliation for new subscribed users.

Although the PubSub specification defines a feature to delete items after a specified time span automatically, the automatic deletion of PubSub nodes is not supported. This feature is required for SSNs, and is useful for the host of the XMPP server. When a large number of unused nodes accumulate over the years, the host may delete nodes without notifying the node owners to free memory.

In PubSub items can be stored on a node persistently. PubSub defines only the ability to retrieve all items of one node. However, the Result Set Management extension (RSM) defines an approach to retrieve only a subset of the stored items such as the last 10 published items on the node. The publishers might update their items, but there is not an ability to retrieve the items created or updated in a certain time span. Furthermore, a user is not able to retrieve only new (unretrieved) items. Therefore, a user must subscribe to a node that stores items persistently and the events must be delivered to the offline storage if the user is offline.

Some operations in SSNs are not atomic. For instance, creating a new SSN requires several messages. There is one message per required PubSub node, one message to publish the SSN to a PEP node, and several messages to join the created SSN. Currently, the client must handle errors during the message transmission and server side errors such as connection losses. For instance, there will be inconsistent data if a connection is lost after for example the first three of five messages. The client will only be able to correct this inconsistency if it is able to reconnect to the XMPP server. Unfortunately, there is currently no public XMPP extension for this problem. An extension that defines transactions would solve this problem. The client could for instance send an IQ message containing several nested IQ messages to the server. The server processes these messages. If an error occurs, the server will rollback the changes of each previously executed message and will send an error message to the client. Otherwise, the server will send all results of the executed operations as a single message back to the client.

The XMPP core protocols rely on a persistent TCP connection between the client and the server. If a connection loss occurs, the connection will be terminated immediately and the client must connect again. Each successful connection attempt involves many messages. For instance, the authentication process, the handshake for securing the streams, the resource binding and the time leveling are repeated. Moreover, the roster, the SSN application list and the node subscription list are retrieved again. Some messages can be saved by caching. However, inconsistencies occur when using the same login on more than one device. A version control approach or an approach similar to the Entity Capabilities [XSF08a] as described in Section 3.4.4 could be used to solve this problem. The Roster Versioning extension XEP-0237 [XSF11b] defines such an approach for retrieving the roster. Schuster et al. [SKS+10] review some approaches for continuing server sessions. The Bidirectional-streams Over Synchronous HTTP (BOSH) XEP-0124 [XSF10a] extension emulates a TCP connection by a synchronous HTTP connection. BOSH is also recommended by Saint-Andre et al. [SAST09] for mobile devices due to the low power consumption. The Stream Management extension XEP-0198 [XSF11c] provides the ability to resume streams by exchanging commands between the sending and the receiving entity.

The roster manages the buddies. Neither the XMPP core protocols nor public XMPP extensions specify

the ability to retrieve the buddies of buddies, or buddies that are more than two degrees away. This feature is commonly used in social networking sites and is favored by most users. However, the query for the buddies of a buddy can be realized by PEP. Each user stores her/his buddies in a PEP node with access model "presence". To retrieve buddies that are more than two degrees away, the access model of the PEP nodes must be "open" or an appropriate access model must be defined. In the first case everybody would be able to see everybody's buddies, which is a privacy issue.

In PubSub there is no user list per node that is visible to each member. Only the owner is able to retrieve the subscription list of a PubSub node. The user list can be stored on another PubSub node, but PubSub does not define an appropriate access model therefor. An appropriate access model would be that only subscribers of a PubSub node have access to another PubSub node.

The in-band registration extension [XSF09b] defines an approach to register a new account over the XMPP network. Whereas the users are able to change their passwords and delete their accounts, a feature to reset a forgotten password is not included. To solve this problem, the XMPP server could send a verification code out of band (e.g., via email) to the user that enables the user to change the forgotten password. Moreover, passwords are stored in plain text by several XMPP server implementations such as ejabberd, because the passwords can be sent in plain text over a secured communication channel. If a person hacks the server or gets access to a backup of the underlying database, the person knows the password of each registered user.

XMPP was a good choice for a prototype implementation, because XMPP solves many low-level problems, there are many useful extensions, several server implementations and libraries readily available and the messages are human readable (i.e., debugging is easy). However, for a release version of SSNs substantial changes to XMPP are required to for example keep the stored data consistent when connection losses occur and to secure SSNs.

7.4 Privacy

The SSN users enter only a user name and a password during the registration process. Thus, the user is only identified by a user name, which is a nickname. The real name is not requested or stored on the server. The available privacy models control the spreading of the user name and the attribution from published content to users. The data of an SSN is not stored on the server persistently, because each SSN has a limited life span. However, SSN applications are able to store the SSN data indefinitely or to send them to a server over the internet. Therefore, the users should be skeptical of SSN applications that own the permission to access the internet on Android.

The buddy based SSN discovery approach shares all SSN applications and SSNs of a user with all buddies of the user. Even though the SSN creator or the corresponding SSN application is able to define if the SSN is discoverable via buddies, the user has no control which SSNs are shared to which buddies. However, a user is able to disable the buddy based discovery. In this case SSNs and applications are not visible to the buddies.

The semi-anonymous access model does not protect the users from scenarios where the owner spreads the user names via other communication channels. For instance, in the YouTube Shoutwall application the user name is visible for every person that is able to look at the shoutwall. Thus, users should consider the application's purpose and the trustworthiness of the SSN creator.

Chapter 8

Summary

This diploma thesis introduces the concept of Spontaneous Social Networks (SSNs), our version of ad-hoc social networks. We present the design of the used software on top of the Extensible Messaging and Presence Protocol (XMPP). We implement a prototype of the SSN framework and three example applications, which demonstrate the capabilities of SSNs. These software components are evaluated in terms of usability with the help of the System Usability Scale (SUS). Moreover, we discuss the scalability and suitability of XMPP and privacy issues.

SSNs are small, well-defined and short-lived social networks. They enable spontaneous social interactions and are used for collaboration, exchanging content and media, notifying an interested audience, data acquisition, dating, gaming, etc. On top of these SSNs applications, which operate on behalf of the users, are implemented. Therefore, the SSNs are tightly coupled with the corresponding applications. The applications define the properties of the SSNs depending on the application's scope or let the users define some or all properties. The properties are the title, description, language, access model, privacy model, time to live (TTL) and the communication model. They are defined during the creation of the SSN. The access model restricts the access to the SSN. The privacy model defines who is able to see the publisher of the messages. The TTL specifies the deletion date of the SSN. The communication model defines in which direction or in which directions the communication takes place.

SSNs are created by users. Each SSN has a unique randomly generated address. Therefore, the SSN addresses are not easily guessable by users. A common problem in SSNs is the discovery of new SSNs. SSNs can be discovered via invitations, similarity matching, visual codes, tones and the buddy based discovery approach. In the discovery approach via similarity matching a user specifies properties of desired SSNs. The properties are location, interests, activity and time. These properties are matched against the properties specified by the SSN creator during the SSN creation. Suitable SSNs are returned to the user. Whereas the discovery via visual codes encodes the SSN application and the SSN address by QR codes, the discovery via tones encodes the SSN application and the SSN address by tones. We define the mapping from symbols to tones and the recognition approach based on the FFT in our reference implementation. The discovery approach via buddies enables a user to discover the SSN applications and SSNs of the user's buddies.

We use a centralized architecture on top of XMPP to realize the message exchange in SSNs. An internet connection is required to use the Publish-Subscribe (PubSub) paradigm of SSNs. PubSub relies on nodes that are responsible for the message exchange. Depending on the SSN's scope and the SSN's properties, we use one or several PubSub nodes. There are SSNs where messages are distributed to every SSN member (i.e., N to N communication model), SSNs where messages are only delivered to the SSN

creator (i.e., N to 1 communication model) and SSNs where messages are only sent from the SSN creator (i.e., 1 to N communication model).

The SSN framework bundles the functionality used in SSNs and offers useful features such as a buddy list, settings and a notification system. It is capable of creating, discovering and managing SSNs. The SSN applications installed on a single device bind to a single instance of the SSN framework. Whereas the SSN approach is not tied to any specific platform or programming language, our implementation is mainly based on the mobile platform Android. Our three implemented SSN applications demonstrate the capabilities and benefits of SSNs. The Central Notifications application enables a user to send text messages to an interested audience. The YouTube Shoutwall enables users of an SSN to post text messages and YouTube videos on a shoutwall. Furthermore, the users are able to vote for or against the current played video. The Event Scheduling application enables users to find a date for an event collaboratively. All users are able to suggest dates and vote for them. It provides suggestions for the best date, and allows users to chat in order to discuss and fix a date.

XMPP is in principle suitable for the message exchange in SSNs. It solves the low-level messaging issues. However, the PubSub extension has some limitations, security and privacy issues concerning SSNs. Therefore, we patched and extended the PubSub specification.

We performed one usability test (System Usability Scale - SUS) with a beta version and one usability test with the final version of our prototype implementation. The result of the beta version/final version is 64.58/80.83 of 100 points. Whereas the beta version test yields an adjective rating based on Bangor et al. [BKM08, BKM09] of "ok", the adjective rating of the final version is "good". The result of the final test lies above the 4th quartile of 206 SUS tests performed by Bangor et al. That is, our prototype implementation belongs to the better products [BKM08]. The accuracy of these results is 35% and 75% based on Tullis and Stetson [TS04].

Chapter 9

Conclusions

SSNs are small, well-defined and short-lived social networks. They enable spontaneous social interactions and are used for collaboration, exchanging content and media, notifying an interested audience, data acquisition, dating, gaming, etc. SSNs can be used on meetings, on events, on weddings, on journeys, in the every day work life, in the leisure time, etc. to make the life of people easier. On top of these SSNs applications, which operate on behalf of the users, are implemented. SSNs are more flexible than ordinary social networks, because they manipulate the underlying social graph without manual intervention. Application developers are able to implement SSN applications by using the offered SSN framework and take advantage of SSNs to build appealing and sophisticated applications. Users can be part of more than a single SSN and achieve different actions at the same time without the notice of other users.

Most people carry their mobile devices such as smartphones always with them. Mobile devices are used to augment the reality, collaborate and communicate with other people. They provide rich communication features. SSNs take advantage of the capabilities of mobile devices. For instance, the internet is used as a communication medium, the user's location is used for the similarity matching SSN discovery, the built-in camera is used for the SSN discovery via visual codes and the microphone is used for the SSN discovery via tones.

SSNs provide suitable access models and discovery approaches for colocated users and users that are acquainted with each other. They enable users to authenticate and join SSNs easily with the help of the buddies access model, the buddy based discovery approach, and the discovery via visual codes and tones. Each discovery approach has several advantages and disadvantages. The SSN discovery via visual codes requires intervisibility between SSN member's devices. The SSN discovery via tones requires the ability of the user's device to record the tones. These discovery methods provide different scenarios in terms of security and range.

The access models and the privacy models secure SSNs and maintain the privacy of the SSN members. SSNs are deleted after maximal 7 days from the creation date onwards. Therefore, the SSN data is not stored on the server longer than 7 days. Whereas SSNs are mainly designed for realtime communication, they are able to store data and deliver messages to users that are offline at the moment the message is sent.

SSNs do not depend on the SSN creator. The SSN creator does not have any special rights except the definition of the properties during the creation of the SSN. Depending on the communication model, the SSN creator can be the only person that is able to receive or send messages.

The performed usability test (System Usability Scale - SUS) of the final version of our prototype implementation yields 80.83 of 100 points. That is, our prototype implementation belongs to the better products [BKM08].

Appendix A

XMPP Message Exchange

Initialization of an XMPP connection and SASL authentication

The following listing shows the initialization of an XMPP connection and SASL authentication. The messages to the server and the messages to the client are sent over an own XML stream. For easier understanding, these streams are merged. Line 3 to 10 show the stream initiation of the client and the server's response. In line 13 to 25 the server sends the available authentication methods and its capabilities to the client and the client sends the chosen authentication method to the server. Line 28 to 49 show the SASL handshake. The server sends two Base64 encoded challenges to the client and the client sends two Base64 encoded responses to the server. Line 48 and 49 show the client's response. In line 52 and 53 the server informs the client that the authentication was successful. Line 56 to 76 show the resource binding and the initialization of the XMPP session. In line 78 to 82 the client retrieves the roster. In line 84 to 87 the client sends the initial presence to the server including the client's capabilities. In line 89 and 99 the server sends the roster to the client. [SA04a]

```
1<?xml version="1.0"?>
2<!-- Sent to server -->
3<stream:stream
4     xmlns="jabber:client"
5     xmlns:stream="http://etherx.jabber.org/streams"
6     to="example.com"
7     version="1.0">
8
9<!-- Sent to client -->
10<stream:stream xmlns="jabber:client"
     xmlns:stream="http://etherx.jabber.org/streams" id="1717318648"
     from="blindeshendl.cg.tuwien.ac.at" version="1.0" xml:lang="en">
11
12<!-- Sent to client -->
13<stream:features>
14   <starttls xmlns="urn:ietf:params:xml:ns:xmpp-tls"/>
15   <mechanisms xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
16      <mechanism>DIGEST-MD5</mechanism>
17      <mechanism>PLAIN</mechanism>
18   </mechanisms>
19   <c xmlns="http://jabber.org/protocol/caps" hash="sha-1"
        node="http://www.process-one.net/en/ejabberd/"
        ver="TeHoUYyKw5MAbWN339k8noZFt2Q="/>
20   <register xmlns="http://jabber.org/features/iq-register"/>
```

```
</stream:features>

<!-- Sent to server -->
<auth mechanism="DIGEST-MD5" xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
</auth>

<!-- Sent to client -->
<challenge xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
bm9uY2U9IjU5NjM2MzI4NyIscW9wPSJhdXRoIixjaGFyc2V0PXV0Zi04LGFsZ29yaXRobT1tZDUtc2Vzcw==
</challenge>

<!-- Sent to server -->
<response xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
dXNlcm5hbWU9InNzbjEiLHJlYWxtPSJibGluZGVzaGVuZGwuY2cudHV3aWVuLmFjLmF0IixjbmU9uY2U9ImFmY
mUxMmZlODVlZjE3ODZkNTBjZWNiMzdmZGEwMTAyOGU1ZDgxOWM3MjBlNzkyMjc2OTAzMmQ1MmM1YTE5NTMiLG
5jPTAwMDAwMDAxLHFvcD1hdXRoLGRpZ2VzdC11cmk9InhtcHAvYmxpbmRlc2hlbmRsLmNnLnR1d2llbi5hYy5
hdCIscmVzcG9uc2U9MWQzYTM4MjgyYzIwOTc2ZmI3MjU0MzY0ZDc5MzU2ZDYsY2hhcnNldD11dGYtOCxub25j
ZT0iNTk2MzYzMjg3Ig==
</response>

<!-- Sent to client -->
<challenge xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
cnNwYXV0aD1jOWIyMDRiYjI3M2E0YWY3MGY1Mzc4MjQ3YTNjNWQ2Zg==
</challenge>

<!-- Sent to server -->
<response xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
</response>

<!-- Sent to client -->
<success xmlns="urn:ietf:params:xml:ns:xmpp-sasl">
</success>

<!-- Sent to server -->
<iq id="0I1zK-0" type="set">
   <bind xmlns="urn:ietf:params:xml:ns:xmpp-bind">
      <resource>ssn</resource>
   </bind>
</iq>

<!-- Sent to client -->
<iq id="0I1zK-0" type="result">
   <bind xmlns="urn:ietf:params:xml:ns:xmpp-bind">
      <jid>ssn1@blindeshendl.cg.tuwien.ac.at/ssn</jid>
   </bind>
</iq>

<!-- Sent to server -->
<iq id="0I1zK-1" type="set">
   <session xmlns="urn:ietf:params:xml:ns:xmpp-session"/>
</iq>

<!-- Sent to client -->
<iq id="0I1zK-1" type="result">
</iq>

<!-- Sent to server -->
<iq id="0I1zK-2" type="get">
   <query xmlns="jabber:iq:roster" >
   </query>
```

```
81</iq>
82
83<!-- Sent to server -->
84<presence id="0I1zK-3">
85   <c xmlns="http://jabber.org/protocol/caps" hash="sha-1"
        node="http://code.google.com/p/ssnetworks"
        ver="/9syVivyC7obAWsIZ5TtoVJ/6cI="/>
86</presence>
87
88<!-- Sent to client -->
89<iq id="0I1zK-2" to="ssn1@blindeshendl.cg.tuwien.ac.at/ssn"
     from="ssn1@blindeshendl.cg.tuwien.ac.at" type="result">
90   <query xmlns="jabber:iq:roster" >
91      <item jid="sepp@blindeshendl.cg.tuwien.ac.at" name="sepp" subscription="both">
92      </item>
93      <item jid="ssn2@blindeshendl.cg.tuwien.ac.at" name="ssn2" subscription="both">
94      </item>
95      <item jid="ssn3@blindeshendl.cg.tuwien.ac.at" name="ssn3" subscription="both">
96      </item>
97   </query>
98</iq>
```

Message Exchange in an SSN

The following listing shows stanzas sent and received by a user in an Event Scheduling SSN. Line 2 to 38 show the query of the PEP node of the Event Scheduling application and the returned SSNs. The returned SSNs are the joined SSNs of the user ssn1. In line 40 to 54 a received event of an Event Scheduling SSN is shown. The event represents a suggestion. Line 57 to 78 show the voting of the user ssn1 in an Event Scheduling SSN.

```
 1<!-- Sent message -->
 2<iq id="0I1zK-12" type="get">
 3   <pubsub xmlns="http://jabber.org/protocol/pubsub">
 4      <items node="com.mh.ssn.application.android.eventscheduling"/>
 5      <set xmlns="http://jabber.org/protocol/rsm">
 6         <max>20</max>
 7      </set>
 8   </pubsub>
 9</iq>
10
11<!-- Received message -->
12<iq id="0I1zK-12" to="ssn1@blindeshendl.cg.tuwien.ac.at/ssn"
     from="ssn1@blindeshendl.cg.tuwien.ac.at" type="result">
13   <pubsub xmlns="http://jabber.org/protocol/pubsub">
14      <items node="com.mh.ssn.application.android.eventscheduling">
15         <item id="oh8lpp088oq1o1419hoql6tmev"
                publisher="ssn1@blindeshendl.cg.tuwien.ac.at/ssn">
16            <ssn xmlns="http://code.google.com/p/ssnetworks/wiki/ssn">
17               <title>Our trip to London</title>
18               <address>hio3Fi(p8</address>
19               <application>com.mh.ssn.application.android.eventscheduling</application>
20               <role>none</role>
21            </ssn>
22         </item>
23         <item id="cb5sh8q62669lbktmi9oir7qea"
                publisher="ssn1@blindeshendl.cg.tuwien.ac.at/ssn">
```

```
24            <ssn xmlns="http://code.google.com/p/ssnetworks/wiki/ssn">
25               <title>Business Meeting with John</title>
26               <address>WjklF+p</address>
27               <application>com.mh.ssn.application.android.eventscheduling</application>
28               <role>owner</role>
29            </ssn>
30         </item>
31      </items>
32      <set xmlns="http://jabber.org/protocol/rsm">
33         <first>modification@001291:727152:995203</first>
34         <last>modification@001291:716935:059364</last>
35         <count>2</count>
36      </set>
37   </pubsub>
38</iq>
39
40<!-- Received message -->
41<message to="ssn1@blindeshendl.cg.tuwien.ac.at/ssn"
     from="pubsub.blindeshendl.cg.tuwien.ac.at">
42   <event xmlns="http://jabber.org/protocol/pubsub#event">
43      <items node="WjklF+p_suggestions">
44         <item id="5166FC74796F7" publisher="ssn2@blindeshendl.cg.tuwien.ac.at/ssn">
45            <ssn-eventscheduling
                 xmlns="http://code.google.com/p/ssnetworks/wiki/ssn-eventscheduling">
46               <suggestion>2011-04-10T17:50:48Z</suggestion>
47            </ssn-eventscheduling>
48         </item>
49      </items>
50   </event>
51   <headers xmlns="http://jabber.org/protocol/shim">
52      <header name="Collection">WjklF+p_suggestions</header>
53   </headers>
54</message>
55
56<!-- Sent message -->
57<iq id="0I1zK-24" to="pubsub.blindeshendl.cg.tuwien.ac.at" type="set">
58   <pubsub xmlns="http://jabber.org/protocol/pubsub">
59      <publish node="WjklF+p_voting">
60         <item id="ssn1">
61            <ssn-eventscheduling
                 xmlns="http://code.google.com/p/ssnetworks/wiki/ssn-eventscheduling">
62              <votes>
63                 <vote suggestionid="50BB60D9CFC18">undefined</vote>
64                 <vote suggestionid="5166FC74796F7">undefined</vote>
65                 <vote suggestionid="50BB0DBEE5533">excellent</vote>
66                 <vote suggestionid="50BB0D73C8CB1">ok</vote>
67                 <vote suggestionid="50BB1421E1840">ok</vote>
68                 <vote suggestionid="50BAF0E058F69">excellent</vote>
69                 <vote suggestionid="50BB0C9B303A1">bad</vote>
70                 <vote suggestionid="50BB0C9F85AEE">no</vote>
71                 <vote suggestionid="50BB0CFCA72D6">no</vote>
72                 <vote suggestionid="50BAF2EB5E6F3">excellent</vote>
73              </votes>
74            </ssn-eventscheduling>
75         </item>
76      </publish>
77   </pubsub>
78</iq>
```

Appendix B

SSN API

The following pages contain an excerpt of the SSN API. Only the interfaces `IPCSSNAdmin`, `IPCSSN` and `IPCActionCallback` are presented, because the complete SSN API would go beyond the scope of this diploma thesis. Furthermore, methods that SSN applications are not allowed to call, are not included. `IPCSSNAdmin` is used to create, join, forget (leave/unsubscribe) and retrieve a joined SSN. `IPCSSN` encapsulates all other actions that can be performed on an SSN such as sending messages and setting a transient node listener. `IPCSSNAdmin` is retrieved when binding to the SSN service. `IPCSSN` is retrieved by calling the `getSSN` or `joinSSN` methods. `IPCActionCallback` is used to track most method calls in `IPCSSN` and `IPCActionCallback`. All other action callbacks look similar except that the `actionSucceeded` method has the result as a parameter.

com.mh.ssn.service.android.ssnadmin.remote

Interface IPCSSNAdmin

All Known Implementing Classes:
IPCSSNAdmin.Stub, IPCSSNAdminImpl

```
public interface IPCSSNAdmin
```

Interface for managing and retrieving SSNs.

Method Summary

void	**createAndJoinSSN**(IPCSSNConfig config, List<IPCSSNNode> additionalNodes, String applicationPackage, String[] subscribeNodeIds, List<IPCPersistentNodeListenerEntity> persistentNodeListeners, IPCNotificationCategoryEntity notificationCategoryToCreate, IPCNotificationNodeListenerEntity notificationNodeListener, IPCSSNActionCallback callback) Creates and joins a new SSN.
void	**displayShareSSN**(String ssnAddress, String applicationPackage, int mode) Displays the SSN Share activity in a new Android task to the user or shares the SSN by the specified method (indicated by parameter mode).
void	**displaySSNDetails**(String ssnAddress, String applicationPackage) Displays the SSN Details (activity) in a new Android task to the user.
void	**forgetSSN**(String address, String applicationPackage, IPCActionCallback callback) Forgets an SSN.
void	**getApplicationSSNs**(String applicationPackage, IPCSSNListActionCallback callback) Returns the SSNs of a certain SSN application.
String[]	**getBuddyGroups**() Returns the buddy groups of the user.
String	**getCustomName**(String username) Returns the custom name of a user.
List<IPCDeletedSSNEntity>	**getDeletedSSNs**(String applicationPackage) Returns the deleted SSNs.
void	**getSSN**(String address, String[] additionalNodeIds, String applicationPackage, IPCSSNActionCallback callback) Returns a previously joined SSN.
void	**joinSSN**(String address, String mainNodeProtocol, String[] additionalNodeIds, String[] additionalNodeProtocols, String applicationPackage, String[] subscribeNodeIds, List<IPCPersistentNodeListenerEntity> persistentNodeListeners, IPCNotificationCategoryEntity notificationCategoryToCreate, IPCNotificationNodeListenerEntity notificationNodeListener, IPCSSNActionCallback callback) Joins a new SSN.

Method Detail

getApplicationSSNs

```
void getApplicationSSNs(String applicationPackage,
                        IPCSSNListActionCallback callback)
                        throws RemoteException
```

Returns the SSNs of a certain SSN application.

Parameters:

`applicationPackage` - The callers (your) application package as defined in the Android Manifest file.
`callback` - Callback that delivers the result or notifies the caller if something went wrong. The error messages are localized by the SSN Service. The error codes can be found in `ErrorCode`. Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a `Handler` in the callback methods!

Throws:

`SecurityException` - If the application package is incorrect.
`RemoteException`

createAndJoinSSN

```
void createAndJoinSSN(IPCSSNConfig config,
                      List<IPCSSNNode> additionalNodes,
                      String applicationPackage,
                      String[] subscribeNodeIds,
                      List<IPCPersistentNodeListenerEntity> persistentNodeListeners,
                      IPCNotificationCategoryEntity notificationCategoryToCreate,
                      IPCNotificationNodeListenerEntity notificationNodeListener,
                      IPCSSNActionCallback callback)
                      throws RemoteException
```

Creates and joins a new SSN. The SSN consists of the main node and the given additional nodes.

Parameters:

`config` - The configuration of the new SSN

`additionalNodes` - The additional nodes or null if no additional nodes are required. The additional node IDs must not contain whitespaces nor any of the following characters: _ " ' & \ / : < > @

`applicationPackage` - The callers (your) application package as defined in the Android Manifest file.

`subscribeNodeIds` - The node IDs of the nodes that should be subscribed. A subscriptions means to get notified of new messages passed onto that node. To process the notifications a notification listener or a persistent node listener must be set for the node. In case of a notification listener, the SSN Framework handles the notifications (i.e. displays them to the user). In case of a persistent node listener, the SSN Application is able to receive the node's messages. You must register a (transient) node listener to receive the node's messages. Null in the array indicates a subscription to the SSN's main node. Pass null as parameter to indicate that no subscriptions are required.

`persistentNodeListeners` - The persistent node listeners of the SSN or null if no persistent node listeners are required. There is maximal one persistent node listener per SSN node.

`notificationCategoryToCreate` - The notification category to create or null if no notification category should be created. The notification category is assigned with the created SSN!

`notificationNodeListener` - The notification listener of the SSN or null if no notification node listener is required. There is only one notification listener per SSN! That is, only one node can have a notification listener per SSN.

`callback` - Callback that delivers the result (the created SSN) or notifies the caller if something went wrong. The error messages are localized by the SSN Service. The error codes can be found in `ErrorCode`. Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a `Handler` in the callback methods!

Throws:

`SecurityException` - If the application package is incorrect.

`RemoteException`

---

joinSSN

```
void joinSSN(String address,
             String mainNodeProtocol,
             String[] additionalNodeIds,
             String[] additionalNodeProtocols,
             String applicationPackage,
             String[] subscribeNodeIds,
             List<IPCPersistentNodeListenerEntity> persistentNodeListeners,
             IPCNotificationCategoryEntity notificationCategoryToCreate,
             IPCNotificationNodeListenerEntity notificationNodeListener,
             IPCSSNActionCallback callback)
             throws RemoteException
```

Joins a new SSN. The SSN consists of the main node and the given additional nodes.

Parameters:

`address` - The address of the SSN

`mainNodeProtocol` - The protocol (XML namespace) of the main node of this SSN

`additionalNodeIds` - The additional nodes or null if no additional nodes are required. The additional node IDs must not contain whitespaces nor any of the following characters: _ " ' & \ / : < > @

`additionalNodeProtocols` - The protocols of the additional nodes. The length and order must be equal to the additionalNodeIds array!

`applicationPackage` - The callers (your) application package as defined in the Android Manifest file.

`subscribeNodeIds` - The node IDs of the nodes that should be subscribed. A subscriptions means to get notified of new messages passed onto that node. To process the notifications a notification listener or a persistent node listener must be set for the node. In case of a notification listener, the SSN Framework handles the notifications (i.e. displays them to the user). In case of a persistent node listener, the SSN Application is able to receive the node's messages. You must register a (transient) node listener to receive the node's messages. Null in the array indicates a subscription to the SSN's main node. Pass null as parameter to indicate that no subscriptions are required.

`persistentNodeListeners` - The persistent node listeners of the SSN or null if no persistent node listeners are required. There is maximal one persistent node listener per SSN node.

`notificationCategoryToCreate` - The notification category to create or null if no notification category should be created. The notification category is assigned with the created SSN!

`notificationNodeListener` - The notification listener of the SSN or null if no notification node listener is required. There is only one notification listener per SSN! That is, only one node can have a notification listener per SSN.

`callback` - Callback that delivers the result (the joined SSN) or notifies the caller if something went wrong. The error messages are localized by the SSN Service. The error codes can be found in `ErrorCode`. Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a `Handler` in the callback methods!

Throws:

`SecurityException` - If the application package is incorrect.

`RemoteException`

---

getSSN

```
void getSSN(String address,
            String[] additionalNodeIds,
            String applicationPackage,
            IPCSSNActionCallback callback)
            throws RemoteException
```

Returns a previously joined SSN.

Parameters:

`address` - The address of the SSN

`additionalNodeIds` - The additional node IDs or null if the SSN has no additional nodes

`applicationPackage` - The callers (your) application package as defined in the Android Manifest file.

`callback` - Callback that delivers the result or notifies the caller if something went wrong. The error messages are localized by the SSN Service. The error codes can be found in `ErrorCode`. Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a `Handler` in the callback methods!

Throws:

`SecurityException` - If the application package is incorrect.

`RemoteException`

---

forgetSSN

```
void forgetSSN(String address,
               String applicationPackage,
               IPCActionCallback callback)
               throws RemoteException
```

Forgets an SSN. This method should be called, when the user wants to leave an SSN. The user is automatically unsubscribed from the subscribed SSN nodes. The persistent node listeners, notification listeners and the associated notification category are deleted automatically. If something went wrong (e.g. connection lost), the request will be processed during the next login.

Parameters:

`address` - The address of the SSN

`applicationPackage` - The callers (your) application package as defined in the Android Manifest file.

`callback` - This callback is only used for synchronization purposes! Therefore, only the actionSucceeded() method is called! Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a `Handler` in the callback methods!

Throws:

`SecurityException` - If the application package is incorrect.

`RemoteException`

---

getBuddyGroups

```
String[] getBuddyGroups()
                    throws RemoteException
```

Returns the buddy groups of the user. This method is required for instance during the creation of an SSN to provide the access model "buddies of the creator".

Returns:

The buddy groups of the user

Throws:

`RemoteException`

---

getCustomName

```
String getCustomName(String username)
                     throws RemoteException
```

Returns the custom name of a user. The custom name is set by the user in the buddy list. If the custom name is not set or the two users are not buddies, the user name will be returned.

Returns:

The custom name of a user.

Throws:

`RemoteException`

---

displaySSNDetails

```
void displaySSNDetails(String ssnAddress,
                       String applicationPackage)
                       throws RemoteException
```

Displays the SSN details (activity) in a new Android task to the user.

Parameters:

`ssnAddress` - The address of the SSN

`applicationPackage` - The SSN application's package that is responsible for the SSN

Throws:
SecurityException - If the application package is incorrect.
RemoteException

---

displayShareSSN

```
void displayShareSSN(String ssnAddress,
                     String applicationPackage,
                     int mode)
                     throws RemoteException
```

Displays the SSN Share activity in a new Android task to the user or shares the SSN by the specified method (indicated by parameter mode).

Parameters:
`ssnAddress` - The address of the SSN
`applicationPackage` - The SSN application's package that is responsible for the SSN
`mode` - The share method. One of `SSNServiceConstants.SSN_SHARE_VIA_ALL`, `SSNServiceConstants.SSN_SHARE_VIA_VISUAL_CODES`, `SSNServiceConstants.SSN_SHARE_VIA_TONES`, `SSNServiceConstants.SSN_SHARE_VIA_INVITATIONS`.

Throws:
SecurityException - If the application package is incorrect.
RemoteException

---

getDeletedSSNs

```
List<IPCDeletedSSNEntity> getDeletedSSNs(String applicationPackage)
                                         throws RemoteException
```

Returns the deleted SSNs. The deleted SSNs include all SSNs which are left (forgotten) by the user, which are automatically deleted due to the TTL is over and which change permissions if the access model is "buddies of the creator" (when the SSN creator removes a buddy from her/his buddy list, the buddy has no longer permissions on all SSNs of the creator with the access model "buddies of the creator"). The SSN Applications have to call this method frequently. With the help of this method the SSN Applications are able to clean up their stored data etc.

Parameters:
`applicationPackage` - The callers (your) application package as defined in the Android Manifest file.

Returns:
All deleted SSNs since the last call of this method or null if no SSNs were deleted.

Throws:
SecurityException - If the application package is incorrect.
RemoteException

com.mh.ssn.service.android.ssnadmin.remote

Interface IPCSSN

All Known Implementing Classes:
IPCSSN.Stub, IPCSSNImpl

```
public interface IPCSSN
```

This interface is used to perform actions on an SSN. Use `IPCSSNAdmin.getSSN(java.lang.String, java.lang.String[], java.lang.String, com.mh.ssn.service.android.remote.IPCSSNActionCallback)` if the SSN has been already joined or `IPCSSNAdmin.joinSSN(java.lang.String, java.lang.String, java.lang.String[], java.lang.String[], java.lang.String, java.lang.String[], java.util.List, com.mh.ssn.service.android.notification.remote.IPCNotificationCategoryEntity, com.mh.ssn.service.android.ssnadmin.remote.IPCNotificationNodeListenerEntity, com.mh.ssn.service.android.remote.IPCSSNActionCallback)` otherwise, to retrieve this interface!

Method Summary

String	**addNodeListener**(String additionalNodeId, IPCSSNNodeListener listener) Adds a (transient) node listener to an SSN node.
void	**addPersistentNodeListener**(String additionalNodeId, String namespace, String elementName, IPCActionCallback callback) Adds a persistent node listener on an SSN node.
String	**getAddress**() Returns the address of the SSN.
IPCSSNMetaData	**getMetaData**() Returns the meta data of the SSN.
void	**getPersistentItems**(String additionalNodeId, IPCAllroundPacketListActionCallback callback) Retrieves all messages that are stored on the SSN node.
String	**getTitle**()

	Returns the title (name) of the SSN.
byte	**getUserRole**() Returns the user role of the user in the SSN.
boolean	**hasNotificationNodeListener**() Returns if a (persistent) notification node listener has been set on this SSN.
boolean	**hasPersistentNodeListener**() Returns if a persistent node listener has been set on this SSN.
void	**publish**(String additionalNodeId, IPCAllroundPacket packet, IPCActionCallback callback) Publishes a message onto an SSN node.
void	**publishNotification**(String additionalNodeId, IPCSSNNotificationPacket packet, IPCActionCallback callback) Publishes a notification onto an SSN node.
void	**publishSameId**(String additionalNodeId, IPCAllroundPacket packet, IPCActionCallback callback) Publishes a message onto an SSN node with the user name as a message ID.
void	**removeNodeListener**(String additionalNodeId, String id) Removes a (transient) node listener from an SSN node.
void	**setNotificationNodeListener**(String additionalNodeId, long notificationCategory, IPCActionCallback callback) Sets a notification node listener on the SSN.

Method Detail

hasPersistentNodeListener

```
boolean hasPersistentNodeListener()
                                throws RemoteException
```

Returns if a persistent node listener has been set on this SSN.

Returns:

True if a persistent node listener has been set on this SSN, false otherwise

Throws:

RemoteException

---

hasNotificationNodeListener

```
boolean hasNotificationNodeListener()
                                 throws RemoteException
```

Returns if a (persistent) notification node listener has been set on this SSN.

Returns:

True if a (persistent) notification node listener has been set on this SSN, false otherwise

Throws:

RemoteException

---

addPersistentNodeListener

```
void addPersistentNodeListener(String additionalNodeId,
                               String namespace,
                               String elementName,
                               IPCActionCallback callback)
                               throws RemoteException
```

Adds a persistent node listener on an SSN node. This method should not be called after creating or joining an SSN! However, if the application data of the SSN framework is deleted or the account is used on another device, the persistent listeners have to be set again, because they are only stored in the database on the device.

Parameters:

`additionalNodeId` - The ID of the additional SSN node or null if the listener should be registered at the SSN's main node.

`namespace` - The XML namespace of the messages that are received by the listener on the SSN node.

`elementName` - The name of the XML root tag.

`callback` - Callback that notifies the caller if the operation has been successfully executed or something went wrong. The error messages are localized by the SSN Service. The error codes can be found in `ErrorCode`. Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a `Handler` in the callback methods!

Throws:

RemoteException

---

setNotificationNodeListener

```
void setNotificationNodeListener(String additionalNodeId,
                                 long notificationCategory,
                                 IPCActionCallback callback)
                                 throws RemoteException
```

Sets a notification node listener on the SSN. There can only be one notification node listener per SSN! This method should not be called after creating or joining an SSN! However, if the application data of the SSN framework is deleted or the account is used on another device, the persistent listeners have to be set again, because they are only stored in the database on the device.

Parameters:

`additionalNodeId` - The ID of the additional SSN node or null if the listener should be registered at the SSN's main node.
`notificationCategory` - The category ID of the notifications that are received with the help of this listener.
`callback` - Callback that notifies the caller if the operation has been successfully executed or something went wrong. The error messages are localized by the SSN Service. The error codes can be found in `ErrorCode`. Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a `Handler` in the callback methods!

Throws:

`RemoteException`

---

addNodeListener

```
String addNodeListener(String additionalNodeId,
                       IPCSSNNodeListener listener)
                       throws RemoteException
```

Adds a (transient) node listener to an SSN node. A precondition is that a persistent node listener was set on this node! There can be more than one listener per SSN node at the same time. Do not forget to remove the node listener!

Parameters:

`additionalNodeId` - The ID of the additional SSN node or null if the listener should be registered at the SSN's main node.
`listener` - The (transient) node listener to add

Returns:

The ID of the node listener or null if something went wrong. The ID is required to remove the node listener.

Throws:

`RemoteException`

---

removeNodeListener

```
void removeNodeListener(String additionalNodeId,
                        String id)
                        throws RemoteException
```

Removes a (transient) node listener from an SSN node.

Parameters:

`additionalNodeId` - The ID of the additional SSN node or null if the listener should be registered at the SSN's main node.
`id` - The ID of the (transient) node listener to remove

Throws:

`RemoteException`

---

publish

```
void publish(String additionalNodeId,
             IPCAllroundPacket packet,
             IPCActionCallback callback)
             throws RemoteException
```

Publishes a message onto an SSN node.

Parameters:

`additionalNodeId` - The ID of the additional SSN node or null if the listener should be registered at the SSN's main node.
`packet` - The packet (message) to send
`callback` - Callback that notifies the caller if the operation has been successfully executed or something went wrong. The error messages are localized by the SSN Service. The error codes can be found in `ErrorCode`. Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a `Handler` in the callback methods!

Throws:

`RemoteException`

---

publishSameId

```
void publishSameId(String additionalNodeId,
                   IPCAllroundPacket packet,
                   IPCActionCallback callback)
                   throws RemoteException
```

Publishes a message onto an SSN node with the user name as a message ID. This method enables the SSN application to update the message on a node that stores messages with the limitation that there is only maximal one message per user on the SSN node.

Parameters:

`additionalNodeId` - The ID of the additional SSN node or null if the listener should be registered at the SSN's main node.
`packet` - The packet (message) to send
`callback` - Callback that notifies the caller if the operation has been successfully executed or something went wrong. The error messages are localized by the SSN Service. The error codes can be found in `ErrorCode`. Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a `Handler` in the callback methods!
Throws:
`RemoteException`

---

publishNotification

```
void publishNotification(String additionalNodeId,
                         IPCSSNNotificationPacket packet,
                         IPCActionCallback callback)
                         throws RemoteException
```

Publishes a notification onto an SSN node.
Parameters:
`additionalNodeId` - The ID of the additional SSN node or null if the listener should be registered at the SSN's main node.
`packet` - The packet (notification) to send
`callback` - Callback that notifies the caller if the operation has been successfully executed or something went wrong. The error messages are localized by the SSN Service. The error codes can be found in `ErrorCode`. Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a `Handler` in the callback methods!
Throws:
`RemoteException`

---

getPersistentItems

```
void getPersistentItems(String additionalNodeId,
                        IPCAllroundPacketListActionCallback callback)
                        throws RemoteException
```

Retrieves all messages that are stored on the SSN node.
Parameters:
`additionalNodeId` - The ID of the additional SSN node or null if the listener should be registered at the SSN's main node.
`callback` - Callback that delivers the result or notifies the caller if something went wrong. The error messages are localized by the SSN Service. The error codes can be found in `ErrorCode`. Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a `Handler` in the callback methods!
Throws:
`RemoteException`

---

getMetaData

```
IPCSSNMetaData getMetaData()
                           throws RemoteException
```

Returns the meta data of the SSN.
Returns:
The SSN's meta data
Throws:
`RemoteException`

---

getTitle

```
String getTitle()
               throws RemoteException
```

Returns the title (name) of the SSN.
Returns:
The SSN's title
Throws:
`RemoteException`

---

getUserRole

```
byte getUserRole()
             throws RemoteException
```

Returns the user role of the user in the SSN.
Returns:
The user role of the user in the SSN. One of `SSNConstants.SSN_USERROLE_OWNER` or `SSNConstants.SSN_USERROLE_NONE`.
Throws:
`RemoteException`

getAddress

```
String getAddress()
                throws RemoteException
```

Returns the address of the SSN.

Returns:

The SSN's address

Throws:

RemoteException

com.mh.ssn.service.android.remote

Interface IPCActionCallback

All Known Implementing Classes:

IPCActionCallback.Stub

```
public interface IPCActionCallback
```

Callback that notifies the caller if the operation has been successfully executed or something went wrong. The error messages are localized by the SSN Service. The error codes can be found in ErrorCode. Do not execute lengthy or error prone operations or change the user interface in the callback methods directly. We recommend to use a Handler in the callback methods!

Method Summary

void	**actionFailed**(String reason, int errorCode) This method is called if the execution of the action has failed.
void	**actionSucceeded**() This method is called if the action has been executed successfully.

Method Detail

actionSucceeded

```
void actionSucceeded()
                 throws RemoteException
```

This method is called if the action has been executed successfully.

Throws:

RemoteException

actionFailed

```
void actionFailed(String reason,
                  int errorCode)
                  throws RemoteException
```

This method is called if the execution of the action has failed.

Parameters:

`reason` - The localized reason why the execution has failed

`errorCode` - The error code why the execution has failed. See ErrorCode

Throws:

RemoteException

Appendix C

XML Schema

SSN and SSN Application Invitation

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://code.google.com/p/ssnetworks/wiki/ssn-invitation"
xmlns="http://code.google.com/p/ssnetworks/wiki/ssn-invitation"
elementFormDefault="qualified">

<xs:element name="ssn-invitation">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="application"/>
      <xs:element name="ssn" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="application">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name" type="xs:string"/>
      <xs:element ref="package" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

Notifications/Central Notifications

This XML schema can be used by all SSN applications. However, the Central Notifications application does not specify any buttons.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://code.google.com/p/ssnetworks/wiki/ssn-notification"
xmlns="http://code.google.com/p/ssnetworks/wiki/ssn-notification"
elementFormDefault="qualified">
```

```
<xs:element name="ssn-notification">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="message" type="xs:string"/>
      <xs:element name="package" type="xs:string"/>
      <xs:element ref="buttons" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="buttons">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="button" maxOccurs="2"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="button">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="label" type="xs:string"/>
      <xs:element name="auto-remove" type="xs:boolean"/>
      <xs:element ref="intent"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="intent">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="categories"/>
      <xs:element name="action" type="xs:string" minOccurs="0"/>
      <xs:element name="package" type="xs:string" minOccurs="0"/>
      <xs:element name="class" type="xs:string" minOccurs="0"/>
      <xs:element name="uri" type="xs:string" minOccurs="0"/>
      <xs:element ref="extras" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="categories">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="category" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="extras">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="extra" type="xs:string" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="extra">
  <xs:complexType>
    <xs:simpleContent>
```

```
    <xs:extension base="xs:string">
      <xs:attribute name="key" type="xs:string" use="required"/>
    </xs:extension>
   </xs:simpleContent>
  </xs:complexType>
</xs:element>

</xs:schema>
```

YouTube Shoutwall

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://code.google.com/p/ssnetworks/wiki/ssn-shoutwall"
xmlns="http://code.google.com/p/ssnetworks/wiki/ssn-shoutwall"
elementFormDefault="qualified">

<xs:element name="ssn-shoutwall">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="message"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="message">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="type" default="text">
          <xs:simpleType>
            <xs:restriction base="xs:string">
              <xs:enumeration value="text"/>
              <xs:enumeration value="youtube"/>
              <xs:enumeration value="vote-for"/>
              <xs:enumeration value="vote-against"/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

</xs:schema>
```

Event Scheduling

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://code.google.com/p/ssnetworks/wiki/ssn-eventscheduling"
xmlns="http://code.google.com/p/ssnetworks/wiki/ssn-eventscheduling"
elementFormDefault="qualified">

<xs:element name="ssn-eventscheduling">
  <xs:complexType>
```

```
    <xs:choice>
      <xs:element name="chat" type="xs:string"/>
      <xs:element name="suggestion" type="xs:dateTime"/>
      <xs:element ref="votes"/>
    </xs:choice>
  </xs:complexType>
</xs:element>

<xs:element name="votes">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="vote" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:simpleType name="VoteType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="undefined"/>
    <xs:enumeration value="excellent"/>
    <xs:enumeration value="ok"/>
    <xs:enumeration value="bad"/>
    <xs:enumeration value="no"/>
  </xs:restriction>
</xs:simpleType>

<xs:element name="vote">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="VoteType">
        <xs:attribute name="suggestionid" type="xs:string" use="required"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

</xs:schema>
```

SSN Application for Buddy Based Discovery

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://code.google.com/p/ssnetworks/wiki/ssn-application"
xmlns="http://code.google.com/p/ssnetworks/wiki/ssn-application"
elementFormDefault="qualified">

<xs:element name="ssn-application">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="package" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

SSN for Buddy Based Discovery

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://code.google.com/p/ssnetworks/wiki/ssn"
xmlns="http://code.google.com/p/ssnetworks/wiki/ssn"
elementFormDefault="qualified">

<xs:element name="ssn">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="address" type="xs:string"/>
      <xs:element name="application" type="xs:string"/>
      <xs:element name="role">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="none"/>
            <xs:enumeration value="owner"/>
          </xs:restriction>
        </xs:simpleType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

Appendix D

Changelog

Changes to Asmack

- The reconnection approach of Asmack does not work. Added new connection listeners to let the client reconnect after disconnections.
- Added `manageGroups` method to `RosterEntry`. The sent roster packet contains the user name and all groups of the user. However, in Asmack the groups can only be managed in `RosterGroup` and only one group can be added/removed to/from a buddy at once. The outgoing and incoming messages are not synchronized. Hence, synchronization errors occur when calling the corresponding method several times with the same buddy but different groups.
- XEP-0077 In-Band Registration + XEP-0158 CAPTCHA Forms: Added support for CAPTCHA forms at the in-band registration extension.
- Data Forms XEP-0004: If there is no field type specified the field type is now not checked anymore.
- Added ability to construct a PubSub node without contacting the XMPP server. This is required for receiving offline PubSub event messages. The node meta data is added if the node is looked up via the existing `getNode` method.
- Improved clean up in PubSub: Added clean up for PubSub nodes (remove listeners) and removing nodes from the `PubSubManager's` node map.
- Added rudimentary support for XEP-0202 Entity Time: `Packet` + `PacketProvider`, but no automatic answer on requests
- Added `description`, `language`, `purge_offline`, `notification_type`, `send_last_published_item`, `node_expire`, `meta` to the PubSub Node config
- Added PubSub Node metadata
- Fixed Bug in Data Forms XEP-0004 in the `setAnswer` method with values as a list: `FormField.TYPE_TEXT_MULTI` must also accept list of values.
- Added publisher of the item to the `PayloadItem` and the parsing of the publisher to the `ItemProvider`. See "12.16 Associating Events and Payloads with the Generating Entity" of XEP-0060 for details! Added options nobody, member, none to `ItemReply`

- Added method for returning the item count in `org.jivesoftware.smackx.pubsub.ItemsExtension` (`getItemCount()`)
- Added support for XEP-0059: Result Set Management (`PacketExtension` and `PacketExtensionProvider`) Added rudimentary RSM support for PubSub `getAllItems()`, `getAllItems(int stepSize)`, `getItemCount()` in `org.jivesoftware.smackx.pubsub.PubSubManager`
- Fixed some bugs in PEP. However use `PubSubManager` instead of the PEP classes! Bugs: no items element, parser chain broken (`PEPProvider`), pep event listener does not work
- Fixed SASL incorrect encoding bug (see Beem project -> http://www.beem-project.com/issues/284 and patch 50-fix-sasl-incorrect-encoding.patch)
- Fixed namespace bug in `org.jivesoftware.smackx.pubsub.provider.PubSubProvider` (`IQProvider`): The PubSub namespace was always forwarded to the parsePacketExtension method. Now the parameter is changed to the correct namespace of the packet extension. Because of that the packet extension parsing is working correctly now.
- Fixed get node bug in `org.jivesoftware.smackx.pubsub.PubSubManager`: Obviously there was an assumption that the identity of type="leaf" is always the first tag in the disco#info answer. But that is not always the case (e.g., for PEP nodes). Moreover, if the type is not "leaf" the node is assumed to be a collection node. If the node does not exist, a collection node is returned. No XMPP exception will be thrown if the node does not exist. All these issues are fixed!
- Fixed registration bug: No user name and no password are sent to the server if the method with the map as a parameter is used.
- Fixed cancel existing registration bug: malformed xml tag in `org.jivesoftware.smack.packet.Registration` (</remove> instead of <remove/>)

Changes to ejabberd

- Added support for `pubsub#itemreply` (Associating Events and Payloads with the Generating Entity) as defined in the PubSub extension [XSF10b] and extend the offered options for `pubsub#itemreply` with the following values: nobody (i.e., nobody is able to receive the publisher's JID), member (i.e., all members are able to receive the publisher's JID), none (i.e., users with an affiliation of none are able to receive the publisher's JID)
- Added support for the PubSub node's TTL (`pubsub#node_expire`): Alter database scheme, alter `pubsub#config` and delete nodes with the help of the cronjob module (`mod_cron`) for ejabberd
- Added support for the node's metadata as described in the PubSub extension [XSF10b]: Alter database scheme (node creator), added `pubsub#description`, `pubsub#language` and `pubsub#type` to node config; added `pubsub#meta`
- Bugfix access model roster in PubSub: Allow owner to subscribe to her/his own node

- Fixed "delete an item from a node" (retract) in PubSub: Owners and publishers are able to delete all items. If the publish model is open, all users will be able to delete all items. The fix ensures that only the item publisher is able to delete her/his item. This disagrees with the PubSub specification!
- Fixed problem when publishing items with the same item ID in PubSub: If a node is persistent (i.e., it is able to store items), an item ID is specified in the publish request and an item with this ID already exists on the node, ejabberd does not check if the publisher of the persistent node is the publisher of the publish request. This error case is not mentioned in the PubSub specification! In this case a forbidden stanza error is returned and the persistent item is not updated.
- Bugfix in the implementation of the privacy list extension [XSF07] (mod_privacy_odbc): When blocking outbound presence notifications, presence messages of all types are blocked. Therefore, the presence subscription approach does not work correctly. The implementation of mod_privacy (i.e., the non-odbc version) does not contain this bug. The fix ensures that presence messages with no type or a type of "unavailable" are never blocked according to the privacy list specification.
- Disabled node config in PubSub for PubSub nodes only
- Disabled the modification/management of the affiliations in PubSub/PEP
- Disabled node deletion in PubSub for PubSub nodes only
- Disabled purging nodes in PubSub for PubSub nodes only
- Disabled manage subscriptions in PubSub/PEP
- Announced and removed new and disallowed features of PubSub/PEP for the `disco#info` request.

Bibliography

[Alt11] Palo Alto. Google's Android becomes the world's leading smart phone platform. http://www.canalys.com/pr/2011/r2011013.html, January 2011 (last accessed on February 2, 2011).

[And11] Android. Android Timeline. http://www.android.com/timeline.html, 2011 (last accessed on February 2, 2011).

[Arm10] Joe Armstrong. Erlang. *Communications of the ACM*, 53:68–75, September 2010.

[BE07] Danah Boyd and Nicole B. Ellison. Social Network Sites: Definition, History, and Scholarship. *Journal of Computer-Mediated Communication*, 13(1-2), 2007.

[BKM08] Aaron Bangor, Philip T. Kortum, and James T. Miller. An Empirical Evaluation of the System Usability Scale. *International Journal of Human-Computer Interaction*, 24(6):574–594, 2008.

[BKM09] Aaron Bangor, Philip T. Kortum, and James T. Miller. Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale. *Journal of Usability Studies*, 4:114–123, 2009.

[BMT07] Dario Bottazzi, Rebecca Montanari, and Alessandra Toninelli. Context-Aware Middleware for Anytime, Anywhere Social Networks. *IEEE Intelligent Systems*, 22(5):23–32, 2007.

[BN07] Rashmi Dutta Baruah and Shivashankar B. Nair. EZeeCom: enabling spontaneous social interactions over mobile ad hoc networks. In *IWCMC '07: Proceedings of the 2007 international conference on Wireless communications and mobile computing*, pages 559–564, 2007.

[Bri88] Elbert Oran Brigham. *The Fast Fourier Transform and its Applications*. Prentice-Hall, Inc., 1988.

[Bro96] John Brooke. SUS: A quick and dirty usability scale. In *Usability evaluation in industry*. Taylor and Francis, 1996.

[Bur09] Jesse Burns. Mobile Application Security on Android. *Black Hat USA*, June 2009.

[CCL03] Imrich Chlamtac, Marco Conti, and Jennifer J.-N. Liu. Mobile ad hoc networking: imperatives and challenges. *Ad Hoc Networks*, 1(1):13–64, July 2003.

[Chi08] Childnet International. Young People and Social Networking Services: A Childnet International Research Report. Technical report, Childnet International, 2008.

[Com08] Communities and Local Government. Online Social Networks. Technical report, Communities and Local Government, October 2008.

[Dec11] Christopher Deckers. The DJ Project - Native Swing. http://djproject.sourceforge.net/ns, 2011 (last accessed on May 20, 2011).

[Doo11] Doodle AG. Doodle. http://www.doodle.com, 2011 (last accessed on May 20, 2011).

[Eja11] Ejabberd. Ejabberd - Protocols Implementation. http://www.ejabberd.im/protocols, 2011 (last accessed on March 1, 2011).

[Fac11] Facebook Inc. Facebook. http://www.facebook.com, 2011 (last accessed on May 20, 2011).

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification*. Addison-Wesley Professional, third edition, 2005.

[GKBB09] Ankur Gupta, Achir Kalra, Daniel Boston, and Cristian Borcea. MobiSoC: a middleware for mobile social computing applications. *Mobile Networks and Applications*, 14(1):35–52, 2009.

[Gol08] Jay Goldman. *Facebook Cookbook: Building Applications to Grow Your Facebook Empire*. O'Reilly Media, Inc., 1st edition, 2008.

[Goo11a] Google Inc. The Developer's Guide. http://developer.android.com/guide/index.html, 2011 (last accessed on March 15, 2011).

[Goo11b] Google Inc. YouTube JavaScript Player API Reference. http://code.google.com/intl/de-AT/apis/youtube/js_api_reference.html, 2011 (last accessed on May 20, 2011).

[Gua90] John Guare. *Six Degrees of Separation: A Play*. Vintage Books, 1990.

[Han08] Sandra Hanchard. Hitwise Asia Pacific Social Networking Report. Technical report, Hitwise Asia Pacific, February 2008.

[HKM10] Sayed Y. Hashimi, Satya Komatineni, and Dave MacLean. *Pro Android 2*. Apress, 2010.

[ISO00] ISO/IEC. *ISO/IEC 18004:2000. Informational technology - Automatic identification and data capture techniques - bar code symbology - QR Code*. ISO/IEC, 2000.

[ISO05] ISO. *ISO 2108:2005. Information and documentation - International Standard Book Number (ISBN)*. ISO, 2005.

[Jiv11] Jive Software. Openfire - Protocol Support. http://www.igniterealtime.org/builds/openfire/docs/latest/documentation/protocol-support.html, 2011 (last accessed on March 1, 2011).

[KP11] Martin Kenney and Bryan Pon. Structuring the Smartphone Industry. Is the Mobile Internet OS Platform the Key? Discussion papers, The Research Institute of the Finnish Economy, February 2011.

[KTC09] Tai-Wei Kan, Chin-Hung Teng, and Wen-Shou Chou. Applying QR code in Augmented Reality Applications. In *Proceedings of the 8th International Conference on Virtual Reality Continuum and its Applications in Industry*, pages 253–257, 2009.

[MAMC10] Mehdi Mani, Nguyen Anh-Minh, and Noel Crespi. SCOPE: A prototype for spontaneous P2P social networking. *8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 220 – 225, 2010.

[Mil67] Stanley Milgram. The Small World Problem. *Psychology Today*, 2:60–67, 1967.

[Mof10] Jack Moffitt. *Professional XMPP Programming with JavaScript and jQuery*. Wiley Publishing Inc., January 2010.

[MyS11] MySpace Inc. MySpace. http://www.myspace.com, 2011 (last accessed on May 20, 2011).

[NBF+11] Chris Nesladek, German Bauer, Richard Fulcher, Christian Robertson, and Jim Palmer. Android UI design patterns. http://www.google.com/events/io/2010/sessions/android-ui-design-patterns.html, 2010 (last accessed on June 19, 2011).

[Ope11] Open Handset Alliance Inc. FAQ. http://www.openhandsetalliance.com/oha_faq.html, 2011 (last accessed on February 2, 2011).

[Ora11] Oracle USA Inc. MySQL 5.1 Reference Manual: The Main Features of MySQL. http://dev.mysql.com/doc/refman/5.1/en/features.html, 2011 (last accessed on March 28, 2011).

[Pal11] PalmSource Inc. OpenBinder. http://www.angryredplanet.com/~hackbod/openbinder/docs/html, 2005 (last accessed on March 21, 2011).

[POL+09] Anna-Kaisa Pietiläinen, Earl Oliver, Jason LeBrun, George Varghese, and Christophe Diot. MobiClique: middleware for mobile social networking. In *WOSN '09: Proceedings of the 2nd ACM workshop on Online social networks*, pages 49–54, 2009.

[SA04a] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920 (Proposed Standard), October 2004.

[SA04b] Peter Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 3921 (Proposed Standard), October 2004.

[SAST09] Peter Saint-Andre, Kevin Smith, and Remko Tronçon. *XMPP: The Definitive Guide: Building Real-Time Applications with Jabber Technologies*. O'Reilly Media, Inc., 2009.

[SHSI08] Stephen Smaldone, Lu Han, Pravin Shankar, and Liviu Iftode. RoadSpeak: enabling voice chat on roadways using vehicular social networks. In *SocialNets '08: Proceedings of the 1st Workshop on Social Network Systems*, pages 43–48, 2008.

[SKS+10] Daniel Schuster, István Koren, Thomas Springer, Dirk Hering, Benjamin Söllner, Markus Endler, and Alexander Schill. Creating Applications for Real-Time Collaboration with XMPP and Android on Mobile Devices. In: Paulo Alencar, Donald Cowan, Handbook of Research on Mobile Software Engineering: Design, Implementation and Emergent Applications, IGI Global, 2010.

[SP06] Karim Seada and Charles Perkins. Social Networks: The Killer App for Wireless Ad Hoc Networks? Technical report, Nokia Research Center, 2006.

[SSS10] Daniel Schuster, Thomas Springer, and Alexander Schill. Service-based Development of Mobile Real-time Collaboration Applications for Social Networks. In *First International Workshop on Communication, Collaboration and Social Networking in Pervasive Computing Environments (PerCom) Workshops*, pages 232–237, 2010.

[TS04] Thomas S. Tullis and Jacqueline N. Stetson. A Comparison of Questionnaires for Assessing Website Usability. In *Proceedings of the Usability Professionals Association (UPA) 2004 Conference*, pages 7–11, June 2004.

[Twi11] Twitter Inc. Twitter. http://www.twitter.com, 2011 (last accessed on May 20, 2011).

[WF94] Stanley Wasserman and Katherine Faust. *Social network analysis: methods and applications*. Cambridge University Press, first edition, 1994.

[WM08] Alfred C. Weaver and Benjamin B. Morrison. Social Networking. *Computer*, 41:97–100, February 2008.

[XIN11] XING Inc. XING. http://www.xing.com, 2011 (last accessed on May 20, 2011).

[XSF03] XSF. XEP-0082: XMPP Date and Time Profiles. http://www.xmpp.org/extensions/xep-0082.html, 2003 (last update on May 28, 2003).

[XSF05] XSF. XEP-0114: Jabber Component Protocol. http://www.xmpp.org/extensions/xep-0114.html, 2005 (last update on March 3, 2005).

[XSF06a] XSF. XEP-0160: Best Practices for Handling Offline Messages. http://www.xmpp.org/extensions/xep-0160.html, 2006 (last update on January 24, 2006).

[XSF06b] XSF. XEP-0059: Result Set Management. http://www.xmpp.org/extensions/xep-0059.html, 2006 (last update on September 20, 2006).

[XSF07] XSF. XEP-0016: Privacy Lists. http://www.xmpp.org/extensions/xep-0016.html, 2007 (last update on February 15, 2007).

[XSF08a] XSF. XEP-0115: Entity Capabilities. http://www.xmpp.org/extensions/xep-0115.html, 2008 (last update on February 26, 2008).

[XSF08b] XSF. XEP-0030: Service Discovery. http://www.xmpp.org/extensions/xep-0030.html, 2008 (last update on June 6, 2008).

[XSF08c] XSF. XEP-0158: CAPTCHA Forms. http://www.xmpp.org/extensions/xep-0158.html, 2008 (last update on September 3, 2008).

[XSF08d] XSF. XEP-0222: Persistent Storage of Public Data via PubSub. http://www.xmpp.org/extensions/xep-0222.html, 2008 (last update on September 8, 2008).

[XSF08e] XSF. XEP-0223: Persistent Storage of Private Data via PubSub. http://www.xmpp.org/extensions/xep-0223.html, 2008 (last update on September 8, 2008).

[XSF09a] XSF. XEP-0202: Entity Time. http://www.xmpp.org/extensions/xep-0202.html, 2009 (last update on September 11, 2009).

[XSF09b] XSF. XEP-0077: In-Band Registration. http://www.xmpp.org/extensions/xep-0077.html, 2009 (last update on September 15, 2009).

[XSF09c] XSF. XEP-0203: Delayed Delivery. http://www.xmpp.org/extensions/xep-0203.html, 2009 (last update on September 15, 2009).

[XSF10a] XSF. XEP-0124: Bidirectional-streams Over Synchronous HTTP (BOSH). http://www.xmpp.org/extensions/xep-0124.html, 2010 (last update on July 2, 2010).

[XSF10b] XSF. XEP-0060: Publish-Subscribe. http://www.xmpp.org/extensions/xep-0060.html, 2010 (last update on June 12, 2010).

[XSF10c] XSF. XEP-0163: Personal Eventing Protocol. http://www.xmpp.org/extensions/xep-0163.html, 2010 (last update on June 12, 2010).

[XSF10d] XSF. XEP-0001: XMPP Extension Protocols. http://www.xmpp.org/extensions/xep-0001.html, 2010 (last update on March 10, 2010).

[XSF10e] XSF. XEP-0248: PubSub Collection Nodes. http://www.xmpp.org/extensions/xep-0248.html, 2010 (last update on September 28, 2010).

[XSF11a] XSF. History. http://xmpp.org/about-xmpp/history, 2011 (last accessed on February 3, 2011).

[XSF11b] XSF. XEP-0237: Roster Versioning. http://www.xmpp.org/extensions/xep-0237.html, 2011 (last update on March 16, 2011).

[XSF11c] XSF. XEP-0198: Stream Management. http://www.xmpp.org/extensions/xep-0198.html, 2011 (last update on March 2, 2011).

[Zha10] Mingxin Zhang. Social Network Analysis: History, Concepts, and Research. In *Handbook of Social Network Technologies and Applications*, pages 3–21. Springer US, 2010.

[Zxi11] Zxing Team. ZXing ("Zebra Crossing"). http://code.google.com/p/zxing/, 2011 (last accessed on June 18, 2011).