Seamless Integration of Location-Linked Services for Smartphones

Vom Promotionsausschuss der
Technischen Universität Hamburg-Harburg
zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation

von
Julian Christoph Ohrt

aus
Hamburg

2017

Date of Oral Examination	May 3rd, 2017
Chair of Examination Board	Prof. Dr. Alexander Schlaefer Institute for Medical Systems Hamburg University of Technology
First Examinor	Prof. Dr. Volker Turau Institute of Telematics Hamburg University of Technology
Second Examinor	Prof. Dr. Rolf-Rainer Grigat Institute of Vision Systems, Hamburg University of Technology

German National Library:
urn:nbn:de:gbv:830-88216556

Zusammenfassung

Das Smartphone hat für viele Menschen die Rolle eines privaten Assistenten eingenommen. Es enthält viele persönliche Informationen und wird immer häufiger für private, sicherheitsrelevante Aufgaben benutzt. Gleichzeitig hat ein Smartphone meistens eine permanente Verbindung zum Internet. Die Folge ist, dass private Informationen und sicherheitsrelevante Aktionen immer häufiger über das Internet übermittelt werden. Im Hinblick auf die stetig steigenden Zahlen von Datendiebstählen, Fällen von Spionage und Internetkriminalität, ist dieser Trend sehr bedenklich. Eine effiziente Methode sich zu schützen, ist auf lokale Kommunikationskanäle statt Internetverbindungen zu setzen.

In dieser Dissertation wird das System BLESS (Building-Linked, Expeditious Services System) entworfen, das es erlaubt, gebäude-gebundene Dienste auf Smartphones zu benutzen ohne vorherige Installation über einen App-Store. Beispiele für solche Dienste sind Steuerungen für Heizung und Licht, der private Familienterminkalender oder das Teilen von Gebäudeplänen zum Navigieren. Ziel von BLESS ist es, für Entwickler eine Plattform zu schaffen, mit der es möglich ist, auf einfache Art und Weise gebäude-gebundene Dienste zu realisieren, die ähnliche Funktionalität wie existierende Smartphone-Apps bereit stellen, jedoch komplett ohne Internetverbindung auskommen. Dabei werden automatisches Auffinden von Diensten, Unabhängigkeit vom Betriebssystem, Schutz der Benutzerdaten und eine vertrauenswürdige Kommunikation zwischen Client und Server in den Vordergrund gestellt. Zum Authentifizierung von Client und Server, sowohl zum Sichern der Verbindungen der beiden Kommunikationspartner werden Zertifikate eingesetzt.

Anhand einer prototypischen Umsetzung wurde das entwickelte Design bewertet und auf Funktionstüchtigkeit getestet. Das Design stellte sich im Feldtest generell als geeignet heraus. Der Entwicklungsaufwand für BLESS-Dienste wurde als angemessen eingestuft. Alle Kommunikationsmethoden zwischen Client und Server funktionierten zuverlässig; ebenso das Auffinden von Diensten. Die Nutzung von Zertifikaten zum Absichern von Kommunikation zwischen Client und Server stellte sich jedoch als impraktikabel heraus. Grundsätzlich wurde das Konzept von BLESS als praktikabel und besonders für private Haushalte gut anwendbar eingestuft.

Contents

List of Figures

List of Tables

List of Abbrevations

AP	access point
app	application for smartphones
BLESS	Building-Linked, Expeditious Services System
cf.	latin: *confer* (compare)
CORBA	Common Object Request Broker Architecture
CP	client part (of BLESS service)
DNS	Domain Name System
DNS-SD	DNS Service Discovery
e.g.	latin: *exempli gratia* (for example)
GCM	Google Cloud Messaging
GNSS	global navigation satellite system
ID	identification
IDL	Interface Description Language
ie.	latin: *id est* (that is)
IGMP	Internet Group Management Protocol
IP	Internet Protocol
JSON	JavaScript Object Notation
LBS	location-based service
LLS	Location-Linked Service
m/DNS	combination of DNS-SD and mDNS
mDNS	Multicast DNS
MOS	mobile operating system
openHAB	Open Home Automation Bus (open source project)
OpenPGP	Open Pretty Good Privacy (open protocol specification)
OSM	Open Street Map (open source project)
PGP	Pretty Good Privacy

PKI	public key infrastructure
POI	point of interest
RFC	Request for Comments
RMI	remote method invocation
RPC	remote procedure call
SDK	software development kit
SLP	Service Location Protcol
SOAP	Simple Object Access Protocol
SP	server part (of BLESS service)
SSDP	Service Discovery Protocol (part of UPnP)
TCP	Transmission Control Protocol
TCP/IP	Internet protocol suite of TCP and IP
TTFF	time to first fix
UDP	User Datagram Protocol
UPnP	Universal Plug and Play
VM	virtual machine
XML	Extensible Markup Language
XMT	cross-platform mobile development tool

Chapter 1

Introduction

In recent years smartphones have become very popular for large population groups due to a wide spectrum of available applications (apps) and falling prices [Ric14]. The number of smartphone shipments is increasing steadily and in North American and European countries smartphone penetration is surpassing 50% these years [eMa14]. With GHz-multicore processors, gigabytes of memory, various integrated sensors, and communication interfaces, the user benefit of smartphones is merely limited by available apps and accessories.

Apps are usually installed via so called app stores. In fact, for most mobile operating systems the app store operated by the vendor is the only officially supported method to install new apps. While this makes installing new apps very user-friendly requiring only a few clicks, app stores have two major drawbacks: They offer a vast and unmanageable variety of apps and the availability of apps cannot be restricted to certain user groups. Both the Google Play Store and Apple's App Store offer well over one million different apps each [Sta15]. The user has to manually search through the vast amount of information, only guided by categories, user ratings, and a text based search. Secondly, due to missing access restrictions, app stores are not well suited for hosting private or personalized apps with limited applicability. For example, apps pre-configured with authentication data for sharing the family calendar must only be available for family members; apps providing internal information about a conference do not need to be world-wide accessible. On the contrary, making such apps publicly available may

raise privacy and security issues.

To circumvent missing access restrictions, usually generic apps are offered which connect to a central server authenticating themselves using credentials entered by the user. Via this server the app can acquire private data and access to private resources hosted by a public server, e.g. incoming text messages or photos collection. Often apps also obtain data from the user, examples include contact details and calendar events. At the same time the smartphone hardware can be used to collect private data, e.g. by determining the users position or by recording audio and video data. Large amounts of private data are thus at the disposal of today's smartphones. While this is necessary to make smartphones user-friendly, helpful, and intelligent personal assistants, it must be made certain that private data is not exposed [Goo15c, App15b].

Mobile operating systems enforce security policies to ensure that private resources are not abused by apps. For each used feature – like access to Internet, contact list, calendar, camera, or the geographical position of the user – an appropriate permission must be owned by an app [FCH+11]. However, fine grained permissions, for example, to authorize an app to obtain the user's position only once or restricting Internet access to the local network, are usually not supported. On the contrary, for iPhone apps full Internet access is a basic privilege which does not need to be requested but is owned by every app by design [Hof13]. During installation and usage the user is informed which permissions are demanded. In general, if the user objects, the only option is not to install and use the app.

As a result, many users install numerous apps granting them a wide range of access privileges. Examples of apps requiring a wide range of permissions include the Facebook [Goo15d] and WhatsApp Messenger [Goo15f] apps as well as the app for updating apps from the Google Play Store [Goo15e]. The former two apps have been installed by 1-5 billion Android users, the latter is usually bundled with the operating system and thus present on virtually all Android devices. All three apps are started when the system booted. They can thus be considered as always running. They own the permission to obtain the user's position at any time and read the local contact list. Given these capabilities and assuming the address

book includes the users' postal addresses, it is possible to exactly derive where the users are living and when they are not at home. The Facebook app can even scan the private calendar to find out when the users planned their vacations. The WhatsApp app can listen to every conversation of its user by accessing the microphone. Finally, all three apps have full Internet access.

Considering such widely distributed and powerful apps, alert users may ask questions like: Are my apps transferring private data? Where is the data sent to? Is it safe and does it stay there? What is it used for? However, most users do not ask these questions given that the apps are providing convenient and well-performing services. This results in vast amounts of private data collected by internationally active companies. App stores – which can only be used by registered users – are prime examples of such data sinks. They are aware of each app each user ever installed and how much users paid for them. As described above, their corresponding updater apps can even determine where the users are living. There can be no doubt that this data might impose severe privacy issues and security risks. Being confronted with an increasing number of reported server break-ins and incidents of leakage, theft, and espionage of data from Internet servers [Inf14, PwC14], this is an alarming development.

The situation is growing even more critical with increasing popularity of current smart home systems. These are usually controlled by smartphone apps which cannot merely collect personal data but can also manipulate physical objects, e.g., open garage doors, control shutters, or disarm the house alarm. An additional security risk is the fact, that many smart home systems relay control signals via Internet servers operated by the vendor. On top of that, if a security vulnerability of a certain type of home automation system becomes known, the app store is technically able to determine at which addresses this system is being used, when its inhabitants are not at home, and thus where and when an easy break-in is possible.

1.1 Contributions

This dissertation proposes a new concept for offering apps which endeavors to amend above mentioned drawbacks and uncertainties. The main innovation is to avoid Internet connectivity, both during installation and runtime, but to solely rely on WiFi communication within local networks. At the same time the discrepancy is dissolved of being limited to install off-the-shelf apps from a global, public mass-market instead of being able to install private, personalized applications making the smartphones a real personal assistant. As the apps introduced by the new concept are not installed via apps stores, they are no traditional apps. They are further linked to the WiFi covered area and will thus be called *location-linked services* (LLSs). In this context, the following contributions are made:

- Potential fields of application for LLSs are identified. Also their limitations are discussed. Based on 16 detailed use-cases, requirements are formulated for providing LLSs. Emphasis is put on a convenient user experience while protecting privacy and providing security.
- In order to provide protection against malware and other unwanted services a concept for service identification is proposed. It allows the user to make an informed decision whether a service can be trusted before installing it. At the same time user authentication is supported. Security related services can thus provide different levels of access for authenticated users.
- After analyzing existing technologies which allow autonomous detection of services once users are admitted to the local infrastructure, a light-weight discovery protocol is developed especially suited for LLSs. Based on this protocol a concept is developed which allows installing services that are comparably powerful as conventional app but can be installed in a distributed manner eliminating the need for a global app store.
- A concept is devised for integrating indoor localization technologies. Allowing LLSs to obtain user location and as well as setting up geofences.

- The design of BLESS (Building-Linked, Expeditious Services System) – a system for providing LLSs in buildings – is presented. Based on existing technologies BLESS is implemented providing a platform for platform-independent, location-linked services. The goal is to allow easy creation of these distributed services which can access smartphone capabilities and communicate with their server counterparts.
- In a field test the system is tested and evaluated. Finally, it is compared to the home automation system openHAB (Home Automation Bus). Differences and similarities are discussed as well as potentials for enhancing BLESS, e.g. by integrating features of openHAB.

1.2 Structure of Dissertation

This dissertation is organized as follows: Chapter 2 provides an overview of technologies connected with LLSs. This comprises an introduction to smart home systems in general, and openHAB in specified. Location-linked services are introduced in detail in chapter 3. Here also examples are provided and areas of applications are highlighted. Chapter 4 considers what requirements a system for providing LLSs must fulfill in order to be beneficial for users as well as suppliers. Afterwards chapter 5 proposes a design for the LLSs system BLESS which is design to comply well with these requirements. The implementation of this system is detailed in chapter 6. As proof of concept chapter 7 presents the application of BLESS to a private household. It also explains how openHAB can be used to provide similar services. Chapter 8 evaluates how well both BLESS and openHAB fulfill the requirements from chapter 4 and finally concludes this dissertation.

Chapter 2

State of the Art

This chapter summarizes the current state of technology and research in context of location-linked service for smartphones.

Section 2.1 presents background information about smartphones: what they are capable of, how they are used, and what current issues and limits are. The following sections provide an overview of technologies which help to create a concept for distributing LLSs for limited geographical areas in a localized manner avoiding app stores and Internet access. LLSs are particularly useful in combination with indoor localization technologies. Consequently, section 2.2 provides an overview of the state of the art of such. It also presents map technologies for visualizing geographical location as well as routing information. In order to compare LLSs to smart home applications, section 2.3 outlines the current state of smart home systems in general and describes in detail the smart home system openHAB with which the system developed by this research will be compared in chapter 7. An important feature of LLSs is autonomous service discovery. According protocols are presented in section 2.4. Before using a discovered service, the communication parties have to authenticate themselves in order to enter a secured and trusted communication. Section 2.5 thus summarizes relevant state of the art about secure networking concerning LLSs. Finally, section 2.6 presents current communication concepts usable by servers and mobile clients of distributed systems in general and of LLSs in particular.

2.1 Smartphones

The iPhone, the first smartphone with multi-touch screen was introduced by Apple in 2007 [DB07]. The first Android device followed one year later [Ger11]. In 2013 for the first time more smartphones than conventional phones were sold worldwide [RM14]. Today smartphones are spread very widely. While hardware capabilities and features of current devices are similar, the operating system is the distinguishing attribute.

2.1.1 Mobile Operating Systems

The choice of the mobile operating system (MOS) directly determines which application format can be installed on a device. Unlike for PCs the operating system of smartphones can in general not be changed. Thus, already when buying a smartphone the user implicitly decides which apps they will be able to use.

Despite the prediction of IDC from June 2011 that the MOS market will be almost evenly split between between the 4 largest MOS (Android, Windows Phone, iOS, and BlackBerry OS) by 2015 [IDC2011], today just the two earliest MOS have a market share above 10%, considering the shares in unit shipments. All other MOSs together have a market share of about 3%. The detailed worldwide MOS market shares for the second quarter of 2015 are shown in figure 2.1.

Android	iOS	Windows Phone	BlackBerry OS	Others
82.8%	13.9%	2.6%	0.3%	0.4%

Table 2.1: Market shares of mobile operating system in 2015Q2 (Share in unit shipments) [IDC]

Even though the MOS market is not as badly segmented as it was predicted, it is still necessary to support multiple MOS to reach 90% of all users. While for PCs multi-platform programming is common practice, e.g., using Java or the QT library for C++, for smartphones it is much less frequently used. The main reasons are performance, user interface, and user experience issues as discussed in [1].

Due to its popularity and wide availability, for this dissertation project the Android MOS was used. However, while technical implementations and details may vary, all used concepts and technologies are in general also applicable and available for other MOSs. If not stated otherwise in the following, examples and details are given based on the Android MOS.

2.1.2 Installation of Apps

App is an abbreviation for *application* which is widely accepted and was even admitted into many English dictionaries[1]. In the following *app* is used for applications for mobile devices, whereas *application* refers to programs for conventional personal computers.

Apps are the only way to add new functionalities to a smartphone. Each executable program is an app. After installation it is usually listed in the smartphone's main app list and is thus launchable be the user. However, many mobile operating systems also allow apps to be started when system events occur, e.g. on boot or on connecting a charger.

The installation routine of new apps puts the biggest constraint on smartphone users currently. Whereas users can get software for personal computers from various sources - including CDs, DVDs, or the Internet - and can afterwards install it on any number of devices, they can only obtain smartphone apps via dedicated app stores. Each MOS provider operates its own app store that only offers applications for that MOS. After creating an application for a particular MOS, the developer submits it to the corresponding app store. Each app store has its own admission policies [Goo15b, App15a]. They may reject apps due to violent content or illicit behavior. The exact test policies are usually not published, however. Once an app store approves an app, all users of that MOS can download and install it. While Android offers the possibility to install apps from not officially supported app stores - so-called *unknown sources* - , iOS does not offer this ability, thus confining its user to Apple's app store.

While the app store is convenient for users since they only have one place to look for apps, it is at the same time a drawback. Quickly distributing

[1] http://www.macmillandictionary.com/dictionary/american/app, http://dictionary.cambridge.org/dictionary/british/app

a beta version or a private app which is only meant for selected users is not easily possible. Further, as app stores are ever growing it gets more difficult to find a right app at the right time. The user has to manually search through more than 1.5 million different apps [Sta15], only guided by categories, user ratings, and a text based search. E.g., when looking for a camera which allows to configure the filename of the taken photos and searching for "camera filename" Google's app store returns 176 hits, as of November 2014.

An advantage for companies and developers is the integrated monetization of apps. They do not have to bother about how end users can pay for apps or returning bought apps, as the stores take care of that. A drawback is, if apps are offered in multiple app stores - e.g. the official one and an *unknown source* - with each new version the app has to be uploaded to each store.

2.1.3 Internet Access

Smartphones are very often sold together with contracts including unlimited data plans. Being *on-line* at all times is the normal state. Smartphones have become a synonym for Internet access.

Indeed, there exist apps for which it is very convenient to be usable at any time, e.g., messaging programs. However, MOSs make it very easy and even encourage apps to make use of the permanent Internet connection. This is the exact opposite to conventional personal computers. Until recently, usually firewalls are used to block Internet access for most applications, while access to files, camera and microphone is granted by default.

However, the trend of limited Internet access for desktop applications is on the decrease. Many applications make use of the Internet connection to check for updates, report usage statistic, or to send crash reports. While these features are usually optional, there also exist applications which cannot be used without an Internet connection, e.g. Microsoft Office 365.

Modern MOSs go one step further and do not try to limited Internet access at all. As a result, all apps could transmit any available data to any Internet server. MOSs must thus provide other means to protect user

privacy and to earn user trust.

2.1.4 Trusting apps

As apps may access all available resources of a smartphone, both user data as well as data collected from hardware sensors, it is of uttermost importance to prevent apps from disclosing private data. If private data needs to be shared, user consent is required. Further, it needs to be made certain with whom and for what purpose the data is shared and that the data is well protected at its destination and is not transferred to other entities. In short, the user needs to trust apps.

To gain the users' trust, all major MOSs pursue a two-fold strategy: App stores and app permissions.

App Stores

App stores allow to efficiently control which apps can be installed by users. If there is no harmful app available in app stores, the users are safe. This is the idea the MOSs producers are promoting, e.g. by calling unofficial app stores *unknown sources*, which is synonymous with *untrusted* sources. In fact all major app stores state strict policies and review guidelines, clearly stating under what conditions apps will not be admitted [Goo15b, App15a]. In theory, this approach is sounds plausible, in real live, however, there are a number of shortcomings.

Most important, screening and filtering of admitted apps is not reliable. In 2013 F-Secure determined that 0.1% of all apps in the Google Play Store contained malware [F-S14, p. 27]. There also have been regularly reports of malware which successfully landed in the App Store [Apv14, XWZ14]. In September 2015 it even became known that a manipulated developer environment infects tens or even hundreds of apps [Xia15]. However, once a new threat or malware type becomes known, app stores tend to be fast in removing affected apps. They can even delete apps from all registered devices without user cooperation nor explicit user consent. In the meantime, however, apps may spread quickly as it is very convenient and fast to install new apps.

The second major issue identified by this dissertation is the fact that there is no easy way for users to determine the authenticity of app developers. Even though each app needs to be signed by the app author, MOSs offer no support to view these signatures before installing an app. It is thus not possible for users to make sure that an app is indeed from a certain author; even if the signature of the author was known. Instead only a arbitrary, self-chosen developer name is shown to the users.

Most critically seen, however, is the role of app stores and its operator, which is the vendor of the MOS. First, each user has to register with the vendor usually providing real name and address. In case the user wants to install apps with costs, also banking account details need to be transmitted. Next, the vendor is aware of each app each user ever installed and how much users paid for them. Next to other conclusions, the vendor can use this information to deduce personal preferences and to identify user groups, e.g. family members or employees of a company. Additionally, when installing apps from the app store users are expected to trust the vendor determining which apps are safe.

Consequently, there is no way for users to use a smartphone without putting unconditional trust in the good nature, honesty, and security mechanisms of the vendor of the MOS. Users can hardly limit nor control which data is collected nor can they ascertain what the data is used for. Even for governments it is difficult to control vendors as they are usually internationally active companies. Data can easily be stored in different locations worldwide, exposing it to any number of attacks.

Permissions

The second approach to enhance trust-level between user and app is being achieved by limiting permissions of the app using the permission system of the MOS. By denying permissions for certain features, these feature cannot be used by an app, e.g. accessing the local address book or the user position via GPS. Android 5 provides an impressive list of 151 different permissions[1] iOS's permission system is much more coarse-grained. An official, complete list is not available.

[1] https://developer.android.com/reference/android/Manifest.permission.html

While the concept of permissions is sound and can efficiently help protecting user privacy, current implementations show several deficits [JMV+12]. Following listing of shortcomings are based on Android 4. Newer versions as well as other MOS may already have amended some of these issues.

A major issue diminishing the effectiveness of permission is presenting a list of permissions to the user during install time. While the user is forced to scrolled down to the end of the list before being able to proceed, the list is often long and its content not well understood [FHE+12]. Resulting in desensitize users especially since there are no direct, negative consequences if the list is not read carefully but simply accepted [MVH88, SM94]. Besides, the user has no choice but to accept in order to use the app.

Further, there are no time-restrictions for permissions. An app with permission to access the camera, could technically start videotaping at any time even if the app is running as background service. There are no one-time permissions, either, e.g. to obtain the user position only once. It is also not possible to allow an app accessing a single contact instead of granting access to the whole address book. Finally, the user is not aware when an app is making use of which permission, e.g. when audio data is being recorded. There is neither an indication of current activities in the notification bar nor is a log made available to the user.

Considering reports of privacy issues and disclosure of personal data, it is most notably that Internet access cannot be restricted for iOS apps at all. Android only distinguishes between full and no Internet access. It is thus not possible to restrict apps to communicate only with selected servers. Also other conceivable restrictions, e.g. to limit number, amount, or rate of data transmissions, are not available.

2.1.5 Limitations

While MOSs are trying to protect user privacy, they enforce the usage of huge, publicly accessible, centralized app stores. As a result large amounts of private data is collected which cannot easily be circumvented by users, as app stores constitute the only officially supported method for installing

apps.

Further, app stores are not suited for distributing following three types of apps in a user-friendly and secure manner:

Location–Linked apps

Apps which are meant to be used in a limited geographical area should not be installed via a global app store. For example, a shopping mall could offer an app providing product finder services for customers. While publishing such an app in an app store, does usually not cause any security concerns, it is not practical. Customers would have to know in the first place that such an app exists, they must actively search for it, and then install it.

Sensitive apps

For apps, owned by companies allowing employees to access internal documents, app stores are not a well suited distribution channel. Also apps controlling physical, security related objects, such as doors, shutters, or alarm systems, are sensitive apps. While these apps require users to enter login credentials before private resources can be accessed, making them globally available via app stores still poses a security risk. They could be downloaded by attackers and be reverse engineered to find security vulnerabilities.

Private apps

Private apps need to be even better protected than sensitive apps. Any private app being accessible by unauthorized users poses a privacy or even security threat. Examples include pre-releases of beta versions for a limited group of test users or pre-configured apps containing access data for private resources, like a family calendar.

2.2 Localization

Hansen et al. pointed out that mobile services often either rely fundamentally on the location of its users or may benefit substantially from it

[HWJT10]. While LLSs can imply that its users are close, a more fine-grained localization of users is also beneficial for these types a applications. This section introduces localization related concepts and technologies.

2.2.1 Location-Based Services

As pointed out by Küpper, the term *location-based services* (LBSs) is being used in the field of mobile communications and mobile computing for many years, although there is neither a common definition nor a common terminology [Küp05, p.1]. In general however, LBSs denote applications integrating some type of geographical location, processing it, and making use of it to generate an additional value for a service [SV04, p.10]. Either the location of the user of the LBS, the location of one or several targets, e.g. of close pizzerias, or both may be used. The term *location service*, on the other hand, is usually used for subsystems generating and delivering the raw location data to the LBS [Küp05, p.2]. However, to prevent overloading the term *service*, this terminology is not used in this thesis; instead *localization system* is used.

LBS can be classified into *reactive* and *proactive* LBSs. A reactive LBS needs to be explicitly activated by the user. On invocation it usually requests the current user position from the localization system and performs based on the obtained position an action, e.g. finding the closest pizzeria. A proactive service on the contrary is automatically invoked as soon as a predefined location event occurs. It could, e.g., send a voucher when the user passes by the pizzeria.

2.2.2 Geofences

For describing location events often geofences are being used. The term *geofence* is a coinage of words *geographic* and *fence*. It is a virtual perimeter for a physical geographic area. Geofences are an integral part of current localization systems. Most MOSs offer a geofence API which allows apps to register for events such as entering, leaving, staying in or outside a given geofence. The operating system takes care about permanently observing the current position and issuing notifications for each registered geofence

to the according app. This concept of a location-based notification service was first introduced by Munson and Gupta [MG02].

Next to facilitating implementation of applications, the geofence concept also allows optimizing energy consumption [Bar12]. For example, when all registered geofences are far away the MOS can rely on a coarse-grained positioning technology which usually is more energy-efficient than fine-grained alternatives. Only when the geofences come closer more fine-grained location sensors need to be activated.

A geofence API usually allows the programmer to specify the geofence itself as well as the transition type. In case of Android's Geofence API[1] the former is specified as a circular area of given radius at a given geographical location specified in latitude and longitude. Supported transition types are *enter*, *exit*, and *dwell*, issued when the user enters or exits the geofence, or dwells inside the geofence for a given amount of time, respectively.

2.2.3 Technologies

Localization systems and their technologies in general can rely on a number of physically measurable quantities. Torres-Solis et al. [TSFC10] distinguish six categories: (1) radio frequency waves, (2) photonic energy, (3) sonic waves, (4) mechanical energy (inertial or contact), (5) magnetic fields, and (6) atmospheric pressure.

While there are many research projects for each category and combinations of them, practically only radio frequency waves are used today: GPS which is a subclass of Global Navigation Satellite Systems (GNSS) has become the de-facto standard for outdoor localization [ME06], while for indoor localization WiFi-based approaches are widely used since they avoid cost for specialized equipment used solely for positioning [HT09]. However, also beacon-based systems are becoming more popular as hardware prices are falling.

[1] http://developer.android.com/training/location/geofencing.html

Global Navigation Satellite Systems

Today there are two GNSSs usable (GPS and GLONASS), while two other are being deployed (GALILEO and Beidou). GNSSs are independent of local infrastructures and provide worldwide coverage. Their satellites contain highly accurate clocks and continuously transmit the time signal to the surface of the earth. GNSS timing receivers can receive the signal and use it to calculate its position with an accuracy of a few meters to several centimeters. A more detailed description and technical details are provided by [Heg12]. After a cold start of a GNSS receiver it takes approximately 60 s to obtain a first valid position (time to first fix, TTFF) [Mau12, p. 76].

This time can be considerably reduced by assisted GNSS [Mau12, p. 77]. It employs an additional data link to quickly obtain data which is needed for localization which includes of satellite ephemeris, almanac, differential corrections and timing information. Without assistance this data is received directly from the satellites at a much lower transmission rate.

GNSSs are very widely used and GPS receivers are integrated in most smartphones. While they provide accurate positions outdoors, they cannot be used for indoor applications due to very strong attenuation of GNSS signal caused by building materials [Mau12, p. 76].

WiFi Localization

An additional receiver of radio frequency waves which is integrated in virtually all smartphones are WiFi adapters. Due to a high availability of local WiFi infrastructure and due to a better penetration of building materials by their radio frequency waves, WiFi is a promising technology to be used for localization services [HWJT10].

Fingerprinting based on RSSI values is the prevalent method of using WiFi for positioning achieving accuracies of a few meters [Mau12, p. 64]. This finding is also backed up by Torres et al [TSFC10, p.57]. Fingerprinting works in two phases. During the *off-line phase* (before the system is operational), WiFi scans are conducted at positions throughout the space where the localization system is to be used. All detected WiFi signals usually including a signal strength indicator are stored in a database together with a

ground-truth location of each scan. This database is often referred to as *radio map*, each entry is called a *fingerprint*. During the *online phase* mobile devices again take WiFi scans and perform afterwards a pattern matching algorithm: In general, they use the radio map as lookup table to find the fingerprint which is most similar and thus determine their own location.

Note that Hansen et all concluded from their experiments that when combining multiple fingerprints into a single one while creating radio maps accuracy will dramatically decrease even in a modestly dynamic environment [HWJT10, p. 9]. As a result respective databases may grow very large in size.

A wide variation of fingerprinting techniques has been developed over the last years. For example, research proposes to enhance accuracy and facilitate processing by:

- applying filters to reduce noise and inference[LBR+02].
- constructing radio maps as weighted graphs [HT09].
- favoring adaptive instead of static radio maps [HWJT10].
- using hyperbolic location fingerprinting to counter the problem that mobile devices measure radio signal strength differently [Kjæ11].
- relying on signal strength differences for building radio maps [LZYP13].

Despite the continuously ongoing research, there is still no prevalent WiFi-based indoor localization system available.

Beacons

On the contrary, commercial solutions rely often on beacons for localization. Beacon-based localization techniques rely on proximity sensing and optionally on lateration as detailed by [Küp05, pp.130]. Hence, they essentially apply the same fingerprinting concept as introduced above. However, in this case fingerprints of the beacons instead of WiFi access points (APs) are taken. Beacons usually have a much lower sending ranges, as a

consequence a dense net of beacons must be installed. This results in fine-grained radio maps which even return accurate results if signal-strengths are inaccurate, e.g. when Bluetooth hardware [Mau12, p.84].

A comprehensive benchmark covering every major beacon manufacturer measuring among other things sending range and battery life time is provided by technology company Aislelabs [Ais15].

Other

More localization technologies exist. For example, angulation relies on measuring the angle of arrival of signals [Küp05, pp.138]. If the clients sends a signal which is received by at least three base stations at fixed positions, a three-dimensional client position can be calculated.

Dead reckoning is a very old technology that was already used by Christopher Columbus and which is still used today [Küp05, pp.141]. It uses a known last position together with sensor data to deduce a new position. While Columbus used a compass and a rope with knots to measure direction and speed of his ship, respectively, today accelerometers, odometers, and gyroscopes often integrated into smartphones are used to determine changes in position.

2.2.4 Indoor Localization Systems

This section presents available systems which can determine the location of smartphones inside buildings.

Localization systems can be grouped into two classes: infrastructure-based and infrastructure-free. Infrastructure-based are called systems which require dedicated hardware which needs to be installed for localization. All beacon-based systems fall into this category. Infrastructure-free are considered systems which do not use any infrastructure or infrastructure which is already available, e.g. WiFi APs or GNSS satellites.

Infrastructure-Based

Many commercial products exist that fall into this category.

One example is the indoor positioning system *StepInside* by Senionlab AB from Sweden [Sen15] for smartphones. It combines WiFi fingerprinting, beacon proximity sensing, as well as dead reckoning using the phone's integrated accelerometer, gyroscope, and magnetometer. Another comparable system is *indoors* by indoo.rs GmbH from Austria[1].

Indoor positioning systems that only rely on beacon proximity sensing include *SmartIndoor* by E-Twenty Development Inc. from Canada[2], a system by Signal360, Inc. from the USA[3], and another system is offered by company Kontakt.io from Poland[4].

All presented products offer software development kits (SDKs) which facilitate creating smartphone apps which make use of the provided indoor location. This includes showing maps being able to present the user's position as well as routes to selected destinations.

Infrastructure-Free

Opposed to infrastructure-based approaches, there exist only few options for infrastructure-free indoor localization systems.

First, there exist large databases containing worldwide WiFi radio maps. Examples include Google's geolocation database [Goo16] as well as a collaborative database Wigle[5]. Both provide an API which can be used to query the geographic location of WiFi access points. However, these approaches are not well suited for indoor localization as their accuracy is as low as 50 m to 100 m and they cannot determine the building level users are in.

Other approaches are usually academic projects like *SmartCampusAAU* [HTTA13] or the crowdsourced approach by Laoudias et al. [LZYP13] and cannot be applied easily in own projects.

[1] https://indoo.rs/
[2] http://smartindoor.com/
[3] http://www.signal360.com/
[4] https://kontakt.io/
[5] https://wigle.net/

2.2.5 Indoor Map Technologies for Smartphones

Without a usage for position data localization technologies are useless. One very useful and wide spread use-case is showing the current position on a map. Another relevant use-case for position data is navigation. It requires additional navigational data. Many map engines and frameworks exist. Some only support outdoor maps, some only indoor floor plans, a few try to integrate both. In the following a indoor map technologies are presented which can be applied to smartphones.

A well known product is Google Indoor Maps[1]. It allows building owners to upload their building maps including all level. When users use Google Maps and zoom in into such an enhanced building, the view changes to show an indoor floor plan. As navigational data is not included, there is no option for showing routes to the user.

Also Open Street Map (OSM) supports describing the interior of buildings[2]. It allows to render indoor maps and also to calculate routes both indoors and outdoors.

Similarly is commercial product *MapsIndoors*[3] by danish company MapsPeople A/S. It extends Google Indoor Maps with navigational data also allowing users to display routes both inside and outside buildings.

AskCody is among other things a way-finding tool only for indoors. It applies its own map technology and allows to show maps and routes on smartphones, websites, and digital information boards, e.g. within shopping malls or hospitals.

Next to these dedicated map projects also the previously presented commercial products for infrastructure-based localization provide SDKs for creating and using indoor maps.

2.3 Smart Home

Controlling and automating technicals devices in private homes is a suitable field of application for LLS. The idea of a *smart home* is already around since

[1] `https://www.google.com/maps/about/partners/indoormaps/`

[2] `http://wiki.openstreetmap.org/wiki/Simple_Indoor_Tagging`

[3] http://mapspeople.com/

the 1980s. Even though houses get smarter – e.g., by using motion sensors for outdoor lighting or by integrating alarm systems which monitor unlocked doors – a real smart home, however, which integrates many electronic and electric devices into a single system as envisioned by [FHW85] is still the exception.

The subject area of *smart home* covers today many different branches, including: home automation, intrusion detection, video surveillance, fire detection, patient health monitoring, and entertainment. For each a large amount of research is and has been carried out. This section presents projects, standards, products in the area of smart home which are relevant for this dissertation.

2.3.1 Smart Home Standards

There exist multiple standards which are used by smart home products. Many of these standards were originally developed by a single company; their specifications are kept secret. However, there exist standards for which the technical specifications are available, such that any vendor can develop compatible products. They are called *open standards* in the following. Subsequently three open standards for communication for smart home products are summarized; followed by an enumeration of the closed standard systems.

X10

X10 is an industry standard for communication among electronic devices primarily via power line wiring. It was developed in 1975 by Pico Electronics and changed a lot since its beginnings [Rye99]. Now the X10 also defines a wireless based transport protocol. Even though it offers only low bandwidth, it is still a popular standard.

Insteon

The Insteon protocol [Ins13] for smart home appliances primarily relies on wireless communication between devices. However, it is compatible to the

X10 standard. It was first published in 2000 by companies Smarthome and Insteon.

Z-Wave

Z-Wave is a radio-only network specification for controlling smart home products. Opposed to Insteon which relies on broadcasting, Z-Wave routes messages. The patent [Sho05] covering the Z-Wave standard was filed in 2005 by ZenSys.

Closed Standards

Next to open standards, multi-purpose technologies like Ethernet, WiFi, or Bluetooth are also used for smart home products. However, due to a lack of a standardized application protocol these devices are usually not easily inter-operable.

Additionally, there exist many proprietary, non-public standards. These are in general used only by a single company that offers their own, closed smart home ecosystem. Examples are given in table 2.2. While offered devices and functions as well as prices vary greatly, they require a matching controller to manage the smart home system.

2.3.2 Smart Home Systems

Creating a smart home system from smart home devices requires a controller which is able to manage all devices. For open standards two approaches can be distinguished: Either a dedicated controller computer or a controller software can be used.

Controller Computer

A dedicated controller computer requires integrated hardware components to interface the above listed standards. Since these controllers are in general not standardized, they require customized firmware. It must provide means to program application logic and to control and manage the whole

System	Vendor	Internet address
Loxone	Loxone	www.loxone.com/
MAX!	eQ3	www.eq-3.de/max-heizungssteuerung.html
brightup	brightup	brightup.de/
FS20	ELV	www.elv.de/fs20-funkschaltsystem.html
HomeMatic	ELV	www.elv.de/homematic-hausautomation-smart-home.html
Synco living	Siemens	www.siemens.de/buildingtechnologies/de/de/gebaeudeautomation-hlk/home-automation-system-synco-living/seiten/home-automation-system-synco-living.aspx
xComfort	Eaton	www.eaton.de/EatonDE/ProdukteundLoesungen/Electrical/ProdukteDienstleistungen/ Wohnbau-kleinereZweckbauten/ xComfortFunkinstallation/index.htm
Dynalite	Philips	www.lighting.philips.com/main/subsites/dynalite/projects/smart_home/
Miele@Home	Miele	www.miele-primus.de/media/shop/home/_prospekte/aktionsprospekte/Miele_Home.pdf

Table 2.2: Examples of closed, proprietary smart home systems.
Links accessed: 2015-08-26

smart home system. For all reviewed controllers the firmware was proprietary and closed-source. The complexity of possible configurations is thus bound to the capabilities provided by the firmware offered by the vendor of the hardware.

An overview of a selection of available controller computers is provided by table 2.3.

Controller Software

The alternative approach is to use a multipurpose computer and equip it with hardware interfaces to make it interoperable with the smart home devices. Then any available controller software can be deployed which uses the interfaces offered by the connected hardware components to take control of the smart home system. Again, the complexity of possible configurations is bound to the capabilities of the software controller. Theoretically, software controllers are interchangeable.

Next to research projects [KBY+12, VF02] which aim to provide software controllers for managing devices belonging to different smart home systems, there also exist commercial products, e.g., IP-Symcon[1]. Additionally, there are many open source projects available having the same goal, e.g., Domoticz[2], LinuxMCE[3], Fhem[4], or openHAB.

Besides all differences concerning functions and capabilities, degree of configurability, used programming language, system requirements, and operating system support, all reviewed software controller systems share a common or at least very similar basic structure: They all provide drivers to connect to hardware interfaces which are capable to communicate with smart home devices. Using some kind of configuration menu or scripts, these devices can be controlled via a web interface or smartphone apps. Further, the systems can react to events, like time or user events as well as events triggered by smart home devices. Some software controller systems are even capable to interact and even manage controller computers

[1] https://www.symcon.de/
[2] http://www.domoticz.com/
[3] http://www.linuxmce.org
[4] http://fhem.de

Controller name	Insteon Hub[a]	ISY994i [b]	Home Center Lite[c]	Home-Troller Zee S2[d]	Vera3[e]
Vendor	Insteon	Universal Devices	Fibaro	HomeSeer	Vera
Requires Internet connection	yes	for App only	no	yes	yes
Closed-Source	yes	yes	yes	yes	yes
Need to register	yes	for App only	no	yes	yes
Configure via	App	Browser	Browser	Browser	Browser
App offered	yes	yes	yes	yes	yes
By default reachable via Internet	no	no	no	yes	yes
Supports X10	no	yes	no	yes	no
Supports Insteon	yes	yes	no	yes	no
Supports Z-Wave	no	no	yes	yes	yes
Developer API offered	yes	yes	no	yes	yes
API requires registration	yes	no	-	no	no

Table 2.3: Selection of available controller computers

[a] http://www.insteon.com/insteon-hub/
[b] https://www.universal-devices.com/residential/isy994i-series/
[c] http://www.fibaro.com/de/the-fibaro-system/home-center-lite
[d] http://www.homeseer.com/home-controllers.html
[e] http://getvera.com/controllers/vera3/

if those provide appropriate programming interfaces. As example, section 2.3.4 provides details about the structure and configuration procedure of openHAB.

2.3.3 Limitations

The major difficulty for a user is the large amount of standards and different systems. Deciding which system is most appropriate for the personal use-cases, while still having a manageable complexity and being within the limits of the financial resources, is a cumbersome task. Research [KBY+12] and economy has detected this problem and reacted by creating alliances aiming to make devices and thus systems inter-operable. Unfortunately, there are again multiple. Examples include: EnOcean alliance[1] and the Qivicon alliances[2].

An alternative way to handle fragmentation of the smart home market are software controllers as introduced in the previous section.

Another problem is that many vendors design their systems based on controller computers in such a way that Internet connectivity is mandatory for a correctly working system. Resulting privacy, security, and connectivity issues are ignored or claimed to be solved. Only the *Home Center Lite* does not require Internet connectivity (cf. section *Controller Computer* on page 23).

None of the systems presented in section 2.3 offers a concept for authentication and authorization of multiple users. Every authorized user is allowed to perform all available actions and read all available data. While for a private house-holds this is usually acceptable, for smart home systems in larger buildings it is not. Neither it is for LLSs.

2.3.4 openHAB

This section introduces the smart home system openHAB. It is used as reference for determining capabilities of LLSs in the field of smart home applications in chapter 7.

[1] https://www.enocean-alliance.org/
[2] https://www.qivicon.com/

The openHAB[1] project is a vendor and technology agnostic open source automation software for private homes which is usually executed on a central server computer. openHAB stands for *Open Home Automation Bus*. It allows switching and dimming of lights, controlling shutters and simple multimedia devices, and can be programmed to react to external events, e.g., weather forecasts and motion alarms. It is extensible so that any event and data sources can be integrated and new hardware devices can be managed and controlled. Software openHAB provides a flexible logging mechanism which can be used to persist all received inputs and performed actions.

openHAB is meant to manually and automatically control home devices. It is accessed via a list-based web browser interface which also supports HTTPS. Additionally, Android and iOS smartphone apps are offered which make use of the same web server interfaces. Usually one home automation device is represented by several list items, displaying the device's state via icons and text and providing according control elements like sliders and buttons. It is even possible to provide customized views for control elements. However, it is not possible to create advanced graphical user interfaces, e.g. for displaying and interacting with maps or dynamic lists. Neither are client-side actions – such as accessing the camera or playing audio files – supported. No support for smartphone-triggered actions such as geofence-events is provided.

openHAB uses a custom, Java-based language for configuring the smart home system. The configuration for each function group is usually stored in a separated file.

First, so called *items* have to be defined in a dedicated configuration file. Items are similar to variables. When setting an item, actions may be triggered, e.g. executing a light switch command. Items are available on the eponymous bus offered by openHAB. Second, scripts (called *rules* in openHAB speech) have to be programmed which respond to item changes, e.g. by changing other items or executing other commands, like issuing an HTTP GET request. Additionally, modules can be configured. They are Java class files which can be loaded into the openHAB framework. Technically, openHAB is a OSGi framework and mentioned class files are OSGi bun-

[1] http://www.openhab.org/

dles. Modules can implement any program logic only limited by capabilities offered by Java, including accessing hardware components which in turn can be used to access smart home devices. Both rules as well as modules can read from and write to any item available on the openHAB bus.

2.4 Discovery Protocols in IP Networks

Discovery protocols are used to detect available services and devices within networks without or with little prior configuration. Many organizations have developed such protocols in the years around the turn of the millennium. Today only four protocols are actively maintained and used. Their specifications are public and all of them have in common that they rely at some point on IP multicast, which is explained in section 2.4.1.

Each protocol distinguishes between the clients which search for items, the items which offer some service, and an optional directory which collects and lists all available items and their services. For easier comparison in the following these three entities will be simply called client, service, and directory, respectively. An overview of their interaction is depicted in figure 2.1.

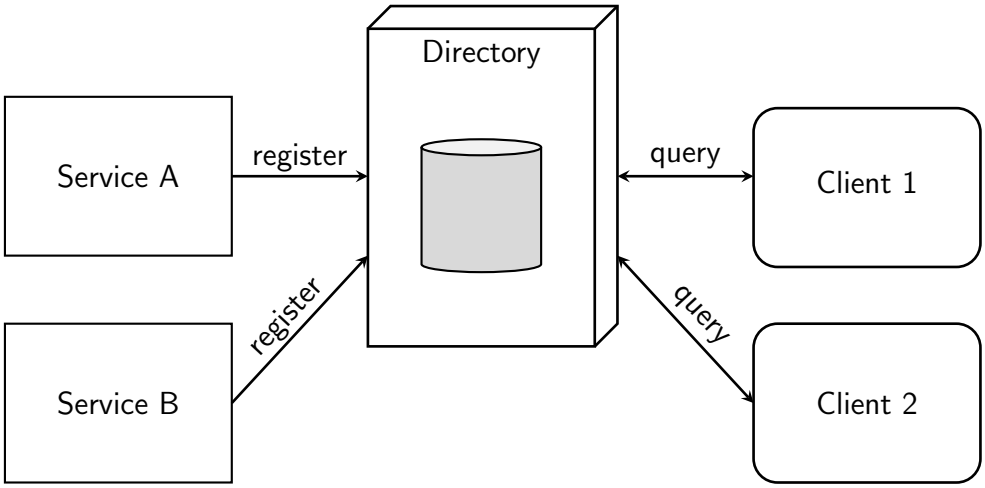


Figure 2.1: Entities involved in service discovery

2.4.1 IP multicast

Multicast describes a one-to-many communication without the need for the sending party to send its message multiple times. Multicast for IP-based networks is defined by number of RFCs, RFC 1112 [Dee89] being the first. IP Multicast is explained in great detail by [Wil00]. This book is used as reference for this section.

Receivers of multicast packets have to register with a multicast address. RFC 1112 reserves IP range from 224.0.0.0 to 239.255.255.255 for this usage. Assignment of multicast addresses is controlled by the Internet Assigned Numbers Authority[1]. Many addresses are reserved for applications or protocols, e.g. address 224.0.1.1 is reserved for distributing the current time via the Network Time Protocol. Further, addresses are grouped into blocks. Relevant for this dissertation are two groups: The *Local Network Control Block* mandates that multicasts in the range of 224.0.0.0-224.0.0.255 are link-local, i.e. their Time-To-Live is set to 1, so that they go no farther than the local subnet. Range 239.0.0.0-239.255.255.255 has been reserved as *Administratively Scoped Block* for use in private multicast domains. It enables administrators to limit multicast scopes within their domains as required.

In order for multicast to work in heterogeneous networks, a series of protocols is required for communication between active network components. First, the Internet Group Management Protocol (IGMP) is used by network clients to register with a multicast address at its network router. Similarly a subscription is canceled later. Multicast messages are then multiplied and send on each active interface – that is those interfaces on which at least one host has an active membership subscription to the according multicast group. Routers that do not support IGMP, have no special treatment for multicast addresses, resulting in unreliable forwarding. For forwarding multicast packets between routers multicast routing protocols exist, such as Distance Vector Multicast Routing Protocol (DVMRP), Multicast Open Shortest Path First (MOSPF), and Protocol Independent Multicast (PIM).

In order for layer 2 network switches to efficiently support IP multicast,

[1] Internet Assigned Numbers Authority (IANA), https://www.iana.org/

the multicast groups must be known. This problem is addressed by protocols IGMP Snooping, Cisco Group Management Protocol (CGMP), and IEEE's Generic Attribute Resolution Protocol (GARP). Switches that do not support efficient multicast forwarding, broadcast according packets instead (multicast flooding).

2.4.2 Current Discovery Protocols

As mentioned above, all current discovery protocols depend to some degree on IP multicast. IP multicast in turn depends on network routers to support IGMP. Switches should avoid multicast flooding. Thus, in order to use IP multicast the network must be configured properly.

Currently maintained discovery protocols are mainly SSDP, mDNS combined with DNS-SD, and SLP. They are summarized in the following sections. A more detailed explanation and comparison is provided by [Sch07].

Further, there exist other projects, e.g., Apache River [Apa13] which is a Java based architecture for building distributed systems. It includes a service discovery protocol which mandates a directory for locating the components of the system. Since it is limited to Java environments and the discovery protocol itself is similar to SLP, it not presented here. Also, there exist open software libraries, e.g. KryoNet[1], which use simple IP multicast for automatic discovery. They are not intended to be used with other libraries, as a result their protocols are usually not well documented and thus not considered in this dissertation.

SSDP

The Simple Service Discovery Protocol (SSDP) is part of the Universal Plug and Play (UPnP) stack. It is publicly specified by the UPnP Forum consisting of more than 1000 companies in computing, printing, and networking [UPn15]. UPnP is widely used as it is integrated by Microsoft – who originally introduced it – into its Windows operating system. The open specification of UPnP version 2.0 was published in February 2015.

[1] https://github.com/EsotericSoftware/kryonet

SSDP uses a multicast group address 239.255.255.250 which belongs to the *Administratively Scoped Block* to discover and announce services. Services are mainly described by a unique service name, a service type, an expiration time, and a location. While the service name includes the type and must never change, the location which is provided in form of a URL may change at any time. Changes are to be announced by the service. Announcements are valid during the time specified by the expiration time. The announcement messages are HTTP formatted, however, are transported using the UDP transport layer protocol.

SSDP devices are discovered by a multicast search. It is possible to discover all devices, a certain service type, or a specific service – in this last case also a unicast search to a given device is possible. Searching by other attributes, like service description or manufacturer, is not supported. All services receiving a discover request answer by sending a unicast UDP reply.

Due to the unreliable nature of UDP, UPnP mandates that UDP packets are sent up to three times. Further, each service needs to be hosted by an device. Devices are arranged in a tree structure where only child devices can contain services. Since all services and devices need to be announced – with each announcement being sent twice – at least six packets have to be multicast to make a single service known in the network.

Implementations of SSDP usually realize the whole UPnP protocol stack. It supports next to service discovery also invocation of service methods and push notifications of service variables to UPnP clients. Both features rely on verbose XML documents which are exchanged between server and client.

DNS–SD and mDNS

Discovery protocols Multicast DNS (mDNS) and DNS Service Discovery (DNS-SD) were introduced by Apple as *Zeroconf* or *Bonjour* (previously called *Rendezvous*). Their specifications are publicly available as RFC 6762[CK13b] and RFC 6763[CK13a], respectively. Apple actively pushed usage of these technologies by introducing them into their Mac operating system beginning with version 10.2 [SC05].

DNS-SD uses Domain Name System (DNS) servers as service directo-

ries. It mandates that a service registers with its DNS server, adding `SRV`, `TXT`, and optionally `PTR` records containing information about its address and service group. Additionally any number of properties can be added. The update procedure is specified by RFC 2136[VTRB97] as *DNS UPDATE*. It requires that the DNS server permits changes to its DNS configuration. Clients query the DNS server for available services using standard DNS queries. Using service groups listing all services of a type is possible.

Apple further introduced DNS Long-Lived Queries as well as Dynamic DNS Update Leases. First enhancement aims to avoid continuous polling for new services by introducing a push mechanism. The later extension makes sure that dynamic DNS entries are deleted from the server if not renewed by the service provider. Currently the only implementation for these extensions are available as package for the open-source DNS server BIND.

Without a DNS server which fulfills this premise, mDNS is used, with *m* standing for *multicast*. It describes how dynamic name resolution can be performed in local networks using IP multicast. With services announcing their own `SRV`, `TXT`, and optionally `PTR` records via multicast, mDNS thus enables DNS-SD in networks without DNS servers. Used multicast addresses belong to the *Local Network Control Block*.

SLP

The SLP (Service Location Protocol) specified by [GPVD99] is the only protocol for service discovery designed by the Internet Engineering Task Force (IETF). It is thus vendor independent and its usage does not require license fees nor a consortium membership.

For small networks SLP supports usage without a service directory. In this case it works similar to SSDP. Client multicast search requests and solicited service answer using unicast packets. Services can also announce their presence using multicast. Used multicast address 239.255.255.253 belongs to the *Administratively Scoped Block*.

For larger installation, a service directory should be used. In this case clients can either multicast their request as before or they directly query the directory for services. For the latter case the address of the directory must be known; clients can either be pre-configured or the directory multicasts

its address periodically to all clients.

SLP services are described by a type, a subtype, and an attribute map. Clients can search for services by type and subtype. A strength of SLP is the advanced query capabilities for attributes allowing to find substrings as well as comparison operators such as '<=' or '>='. A major drawback is its listening port 427, since UNIX system require root access for listening to ports below 1024[1].

2.5 Secure Networking

As detailed by Kruegel et al. in [KVV05, pp. 9], in computer systems in general – and in networking particularly – information flows from a source to a sink over a communication channel. Such a communication is considered secure if only authorized parties according to the security policy in use can access the transmitted data. Figure 2.2 shows the desired information flow and also four categories of attacks.

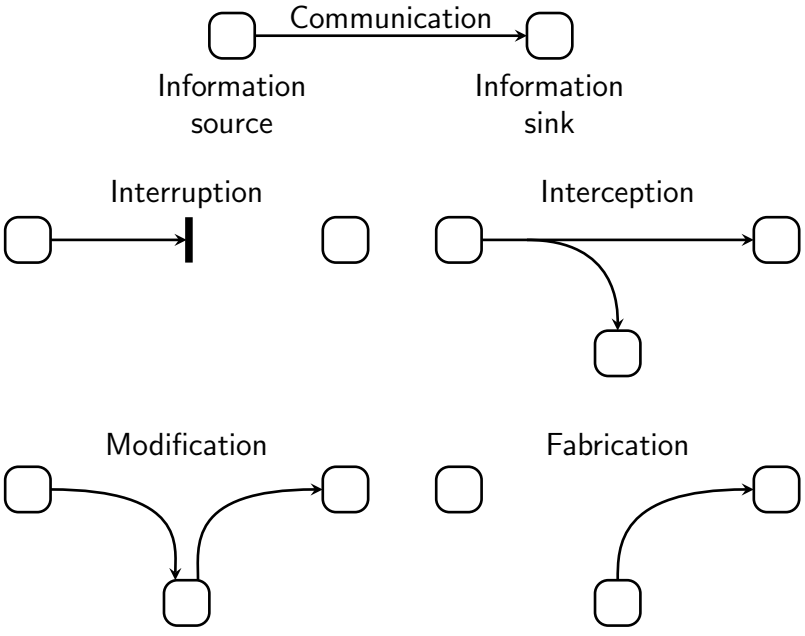


Figure 2.2: Normal communication flow and attack patterns (based on [KVV05])

[1] https://www.w3.org/Daemon/User/Installation/PrivilegedPorts.html

Coulouris et al. pointed out that security for information resources has three properties: confidentiality (protection against disclosure to unauthorized individuals), integrity (protection against alteration or corruption), and availability (protection against interference with the means to access the resources) [CDK05, p. 18].

As visualized in figure 2.3: While interruptions (e.g. by a denial of service attack), violate the availability property, interceptions (e.g. by wiretapping) and modifications (e.g. via man-in-the-middle attack) offend the confidentiality property. Modifications also present an infringement of the integrity property, as do fabrication attacks. Fabrications can either be a replay attack by inserting a previously intercepted message into the communication channel, or it can be masquerading if messages are sent and received using the identity of a third party without their authority.

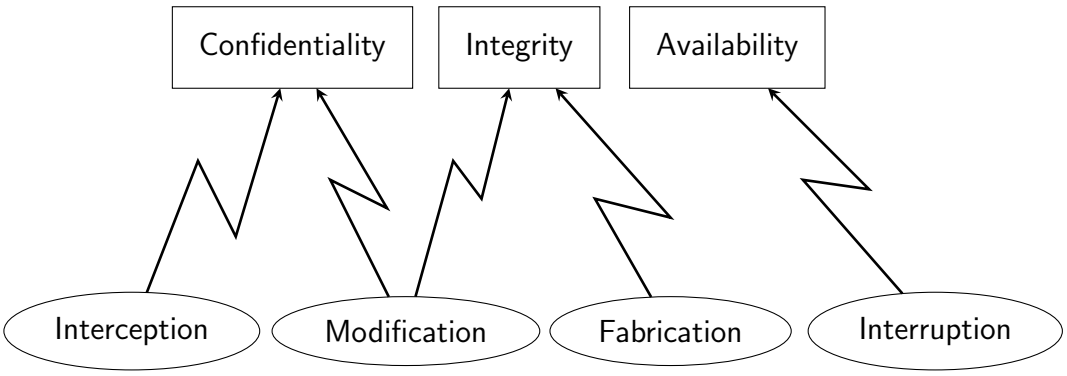


Figure 2.3: Security properties according to Coulouris et al. [CDK05] and attacks patterns

For securing digital communications against eaves-dropping and tampering encryption is used. Modern cryptography also provides mechanisms for protecting against malign repeating of messages and for ensuring the sender's authenticity. Confirming and trusting the sender's authenticity in turn allows digital access control.

2.5.1 Encryption

In order to establish a trust level between two (digitally) communicating entities, they have to prove that they are indeed who they are claim to

be. As of the current state of the art, there are two main classes used as summarized by [CDK05, pp. 473]. They are both based on the use of secrets called *keys*. Cryptographic keys are used by encryption algorithms in such a way that the encryption cannot be reversed without knowledge of a compatible key.

The first encryption class uses *shared secret keys* which must be only known by the entities involved in the current communication. Message are symmetrically encrypted and decrypted using a shared key. The other encryption class is much more complex and was proposed by Diffie and Hellman [DH76]. It is called public-key cryptography since it uses *public/private key pairs*. Each communicating entity owns its own key pair, where the private key is only known by the entity itself and the public key is made available to the public. The sender uses the public key of the intended receiver to encrypt a message. The receiver then needs to provide its private key to the encryption algorithm to decipher the message. Since this asymmetric encryption is much more expensive, it is often used only for sharing a temporary, shared secret key, which is then used for symmetric encryption [KVV05, p. 16].

2.5.2 Certificates

Public/private key pairs also allow to verify that a message or a document is an unaltered copy of one produced by the signer. For digital signing the sender encrypts the message or a compressed form of the message using its own private key. By decrypting the message using the public key of the sender, the receiver assures that the message originated unaltered from the sender. Digital information that was signed is often referred to a certificate. The signer certifies the contained information. The concept of digital certificates was developed by [Koh78].

In today's Internet, certificates are most often used for authenticating websites: A trusted entity signs the address of a website. The created certificate is owned by the website and used to authenticate itself to web users. It can also be used to create further certificates, thus creating a certificate chain. An entity creating certificates is called a *Certificate au-*

thority. Certificate authorities are considered the *public key infrastructure* (PKI) as they are required for starting a certificate chain. Details about SSL and certificates are provided in [Hir97].

Similarly, certificates are also used to authenticate the origin of e-mails. However, instead of relying on a PKI, for e-mailing the so called *web of trust* was introduced. The main idea is that users create their own certificates containing their own e-mail addresses. The public parts of these certificates are handed to other users who check and confirm that contained e-mail addresses indeed belong to the certificate holder. The concept was further described by Philip Zimmermann in 1992 in the manual for PGP (Pretty Good Privacy) version 2.0:

> As time goes on, you will accumulate keys from other people that you may want to designate as trusted introducers. Everyone else will each choose their own trusted introducers. And everyone will gradually accumulate and distribute with their key a collection of certifying signatures from other people, with the expectation that anyone receiving it will trust at least one or two of the signatures. This will cause the emergence of a decentralized fault-tolerant web of confidence for all public keys.

Today the web of trust is well used and defined by the OpenPGP standard (RFC 4880). Next to the original PGP software for signing, encrypting, and decrypting e-mails and files, now also other software is available, e.g. GnuPG[1]. Also multiple public key servers are available which provide a directory of published public keys and keep their databases synchronized, e.g. keys.gnupg.net and pgp.mit.edu.

OpenPGP allows to sign certificates with one of five different levels of trust: I don't know, I do NOT trust, I trust marginally, I trust fully, I trust ultimately. Depending how many signatures and of which trust level a certificate holds, the certificate can be considered validated or not. By default a key K is validated if it meets both of the following two conditions:

1. It is signed by enough valid keys, meaning one of the following:

[1] https://www.gnupg.org/

- You have signed it personally
- It has been signed by one fully trusted key
- It has been signed by three marginally trusted keys

2. The path of signed keys leading from K back to your own key is five steps or shorter.

2.5.3 Digital Access Control

The basic concept of protecting and controlling access to digital resources is described in a classic paper by [Lam71] and summarized by [CDK05, p. 479]. Simplified, for a specific service a server receives a request message containing the operation, which is to be applied, the resource, on which the operation is to be applied, and the identity of the requesting entity. In the first authentication step, the server has to authenticate the requester making sure that they are who they claim to be. During the following authorization step – which is the actual digital access control – any request for which the requesting principal does not have the necessary access right to perform the requested operation on the specified resource is refused.

2.6 Service Communication

Since the beginnings of global networking Internet communication is based on the Internet protocol suite (TCP/IP) which provides two-way end-to-end communication between server and client. However, Internet communication used to purely adhere to the request-response pattern. Specifically client-initiated communication was not envisaged. In fact, before WebSockets[1] and HTTP/2[2] were introduced in 2011 and 2015, respectively, there was no real, browser-based server to client push communication possible.

[1] standardized by RFC 6455 and introduced by HTML5
[2] tools.ietf.org/html/rfc7540

2.6.1 Server to Smartphone Communication

With the introduction of mobile networking clients, e.g. smartphones, providing push messaging became even more complex due to their mobile nature and limited resources. Currently following techniques for realizing push messaging for smartphones exist:

Smartphone becomes server by opening a TCP port and listening to incoming connections at any time. This is the traditional approach for server-to-client-messaging. However, as smartphones do not have a fixed IP addresses, they must distribute it together with the listening port to all communication partners. In case the IP address changes, all partners have to be notified again. Further disadvantages are that communication partners have to retry sending messages if the smartphone is temporarily not available. Also the smartphone must be protected against attackers. The major issue is, however, the network must be configured such that incoming connections to smartphones are allowed. In local networks this is usually given via WiFi connectivity, however, not in cellular networks.

Smartphone keeps open connections to each of its servers. These connections can be used by the servers at any time. If the phone loses connectivity, it has to attempt to reconnect as soon as possible. While this approach works for common network configurations, it is resource consuming as each server needs to hold an open connection for each client. Also the smartphone needs to hold open connections: One for each connected server. Due to keepalive signals frequent network activity is to be expected even if no payload is transferred. High battery drain is inevitable. Further, there exists an ARP protocol issue for servers running Windows operating system and Android clients which entered the so called deep sleep mode. It is discussed in detail by reference [Mil15].

Smartphone keeps a single open connection to a global server. The global server assigns a random identification (ID) to each connected smartphone. A smartphone can pass its ID to third-party servers which in turn can use it to instruct the global server to deliver a message to the smartphone

on their behalf. For bidirectional communication the smartphone needs then to establish a TCP connection to the third-party server.

Today all popular mobile operating systems use this method to support push notification services [Goo15a], [App15c], [Mic15], [Bla14]. These systems are owned by the vendors and integrated into their mobile operating systems. Details about technical aspects, privacy, or security are in general not available. Drawbacks of this method are that the user needs to trust the relay server and needs an Internet connection to reach it. An advantage is easy usage, as it is available and activated on all major mobile operating systems by default.

Smartphone registers for multicast address in local network. Servers can now send their messages to the known broadcast address. A disadvantage of this technique, which is for example applied by UPnP, is that private notifications to a single recipient are not possible. All smartphones listen to all multicasts and must process them. This is resource consuming as it regularly wakes up sleeping devices. It can further cause privacy issues and multicasts must be supported by the network infrastructure (refer to section 2.4.1). Further, in order to detect lost packets a dedicated acknowledgment mechanism has to be introduced.

2.6.2 Smartphone to Server Communication

Considering the method of contacting a network or Internet server, there is no difference between a stationary client or mobile smartphone. Both establish a TCP connection by providing destination host name or IP address and port. Neither the TCP/IP protocol stack nor the networking infrastructure distinguish between the type of client. Thus, smartphones can establish network connections to servers like conventional clients.

2.6.3 Invoking Server Methods

Communication in distributed systems can be divided into three communication paradigms. All of them can be used for invoking methods on remote

servers. This section lists for this dissertation relevant approaches. A detailed description can be found in [CDK05, pp. 145-277].

Interprocess Communication

Communication based on UDP datagrams and TCP connections – the simplest form of interprocess communication – is the base for all other more abstracted communication paradigms. In general, the programmer has to define a protocol that describes the communication data at byte level. Even though this approach is most flexible, it is also most complex and expensive. More details in [CDK05, chapter 4].

Remote Invocation

Remote invocations provide a two-way data exchange between communicating entities which usually resembles synchronous calling of remote methods which respond with a return value. A typical representative is *Remote Procedure Calls* (RPC) described by [BN84].

RPC's main purpose is that procedures in processes on remote computers can be called as if they were procedures in the local address space. Remote method invocation (RMI) is similar to RPC. However as it is object-oriented, it makes whole objects instead of procedures remotely available as if they there locally present.

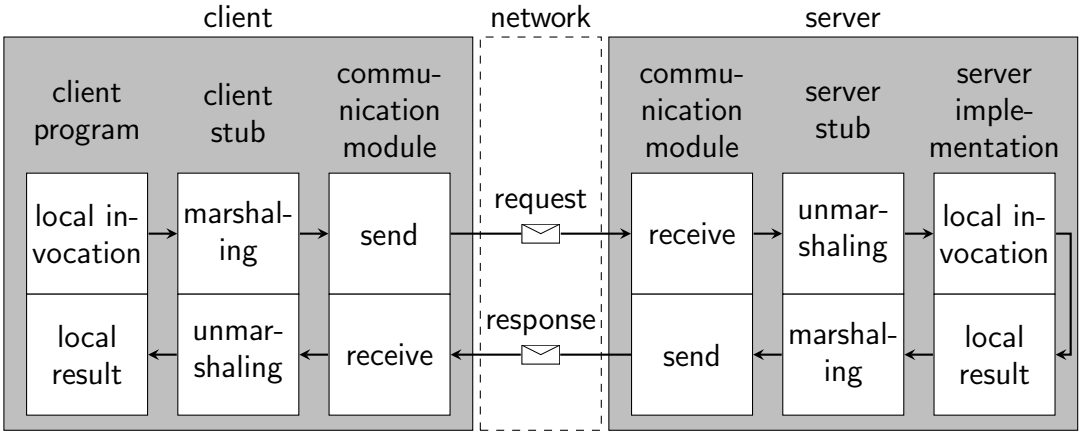


Figure 2.4: The components of a remote invocation system, and their interactions for a simple invocation (based on [BN84])

This transparency is achieved by introducing *stub procedures* and *stub objects* for RPC and RMI, respectively. On the client-side *stubs* behave like local procedures and objects, respectively. But each access request is forwarded via the communication module to the according *server stub*. For transferring invocation parameters - which may be complex object hierarchies using inheritances and pointers -, they are marshaled into serialized form. At the server the parameters are unmarshaled and passed to the corresponding procedure or object implementation. Return values are then passed back to the server stub, marshaled, and send to the client. Finally, the client stub unmarshals the response and passes it to the originally requesting entity. A graphical representation of this process in shown in figure 2.4. More details in [CDK05, chapter 5].

Note that the term *marshaling* as originally introduced by Nelson in the context of RPC only refers "the process of packaging a parameter record into a call or return message" [Nel81]. However, for remote invocations also the method, to which the parameters are to be passed, needs to be uniquely identified. For this reason in this dissertation, *marshaling* will be used for packaging both the parameters together with all necessary information about the called method. For packaging only parameters the term *serializing* will be used, according to current literature.

Indirect Communication

Indirect communication takes abstraction even a step further. Its manifold appliances include publish-subscribe systems, message queues, and distributed shared memory. Details can be found in [CDK05, chapter 6].

Chapter 3

Location-Linked Services

This chapter describes the novel concept of location-linked services (LLSs). First, LLSs themselves are introduced in section 3.1 and motivated in section 3.2. Thereafter section 3.3 outlines a generic system for providing LLSs to users. This chapter further presents use-cases for LLS (section 3.4), highlights difference to related concepts (section 3.5), and lists areas of application for LLS (section 3.6).

LLSs were originally introduced in [2] as *building-linked services*.

3.1 Definition of Location-Linked Services

Location-linked services are distributed applications which are intended to be used within bounded, WiFi covered geographic areas, such as a single room, a private house, an office block, or a university campus[1]. A provider offering LLSs holds both the client and the server application. The server application is executed by the LLS provider. The client application is transferred to a smartphone and executed there. It is capable of providing similar services as native smartphone apps, however by default it is limited to communicate with its corresponding server application. This definition of LLSs is visualized in figure 3.1.

[1] Note that WiFi roaming, i.e. the handover from one AP to the next, is drastically quickened if WiFi standard 802.11r is supported by the network devices.

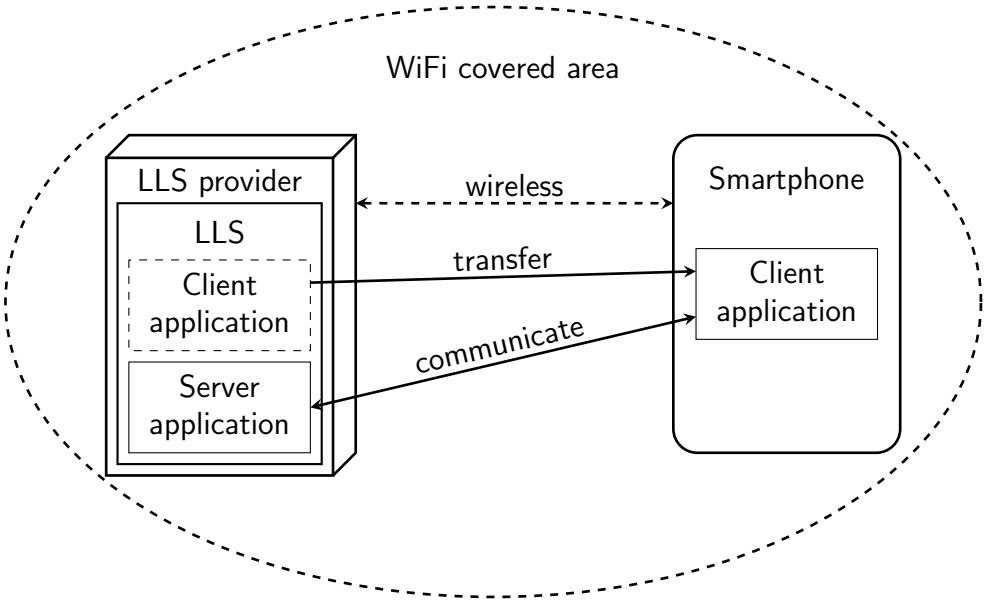


Figure 3.1: Architectural overview location-linked services

3.2 Motivation of Location-Linked Services

LLSs provide some advantages compared to conventional apps in terms of privacy and user comfort. In particular, the introduction of LLSs is motivated by three limitations of current MOSs and their concept of app distribution.

- As detailed in chapter 1, app stores collect large amounts of user data. They can thus be considered omniscient. Most notably, users cannot avoid them as they are the only officially supported method to install new apps (cf. section 2.1.2).
- Apps stores require users to manually search and install apps of interest. However, for apps with limited geographic area of usage (cf. Location-Linked apps in section 2.1.5) it would be more user-friendly, if they were presented proactively to users.
- App stores are globally and publicly accessible. Relying on them for offering sensitive and private apps may foster privacy and security issues as described in section 2.1.5.

LLSs – as defined in the previous section – allow to amend these shortcomings. In order to utilize their full potential a system is required which

allows users to install, use, and manage LLSs.

3.3 Detailed Description of a Generic Location-Linked Services Provisioning System

This section provides a description of a generic provisioning system for LLSs. It also includes resulting consequences which are used to formulate a set of requirements in chapter 4.

Ideally, LLSs do not need to be searched for by the user. Instead when users enter a building offering a location-linked services provisioning system, they are informed about available services. The notification type is configurable by the user. It could be similar to an incoming message notification. Available services may be shown as list entries, to be activated at any time by the user. Services must never be executed without user consent.

Since buildings may offer a great many LLSs, they must be filtered. User, location, and optionally context filters are proposed which allow to hide services not of interest for users at the current point in time. User filters allow LLSs to restrict its services to certain users. This implies that the system provides means for authenticating users. Allowing LLSs to limit its detection range, enables location filters which hide all services which are not intended to be used at the current user position. This in turn requires the provisioning system to provide a localization system to determine the user position. Optionally, context or other advanced filters may be applied, e.g., when just passing through a shopping mall in order to take a shortcut, no notifications about available services should be issued.

LLSs – in particular their client applications – can access sensitive user data. It is thus necessary to prevent users from installing and using malicious or untrusted services. Hence, the provisioning system needs to provide an authentication mechanism which allows services to prove to users their legitimacy and that it is safe to use them. LLSs must also be able to present their benefits to users before installation, e.g. by providing a descriptive text.

To increase trust in LLSs further, the provisioning system should implement a permission system. It needs to allow users to identify permissions services are requesting. The user is to be informed in a non-disruptive manner whenever a service is making use of its permissions. Excessive or unwanted permissions can be denied before they are granted or be revoked at a later time. Services must handle limited access rights gracefully, e.g., by avoiding features which depend on missing permissions. The services system must protect user privacy wherever possible. If user data is shared or published the user should be aware of it and be able to object.

Once all available, trusted, and usable services are found and displayed, users can choose to execute any of them at any time. On first launch of a service, the provisioning system must install the LLS by transferring the client application from the service provider to the smartphone. Afterwards, a started client application behaves similarly to a conventional smartphone app, i.e., it should provide a graphical user interface, be able to access system functionalities and components, and be able to stay in background to be woken later when a specific event occurs. It should also be able to interact with client applications of other services. Communication between the service client running on the smartphone and the service back-end offered by the corresponding server, needs to be bi-directional and secure. Both entities need to be able to send messages at any time. A mechanism for updating services needs to be available.

Services which require to contact their corresponding server application should only be shown to the user while the server is available. However, as the connection could break at any time, the client application and its data files should be installed on the smartphone phone nonetheless. This way the client application can inform the user that the service is temporarily not usable because the server application is unavailable. On the other hand, client applications which never or only rarely require server connectivity, e.g. a navigator service, are always invokable.

3.4 Examples of LLSs

This section lists 16 LLSs which are realizable by the previously introduced generic services system. Some are fictional at this point in time. Many were already discussed in literature. Even though the collection only contains a fraction of conceivable services, it is assumed that a system, which is able to model these presented services, can also be used to create many other, useful services.

All following services can only be installed and used at the location to which the service is linked. If not otherwise stated, linkage to a house is assumed.

3.4.1 Navigation

Similar to GPS-based navigation systems, a navigation service is able to show a map with the current user position. The viewing direction is determined using the compass. Indoor maps with multiple levels need to be supported. Users can choose points-of-interest from a list and calculate paths between multiple positions.

The navigation service can also be invoked by other services. This way other services can easily visually present routes or locations of points of interests without implementing an own map viewer.

3.4.2 Automated Door Bell

Instead of having a visitor physically press the conventional bell button, this service is used to ring the bell. Once the service is installed it detects when the visitor approaches the building and rings the bell just before reaching the door. It is intended for frequent visitors and inhabitants without key.

3.4.3 Audio messenger

Users having installed the audio messenger can send and receive audio messages. If the receiver's phone is not muted and is located inside the building, the announcement is played back as soon as it arrives. Otherwise the sender

is informed that the message is delayed or undeliverable, respectively. For example, this service could be used to personally page a passenger at the airport or an employee needed for assistance.

An enhancement would be a two-way intercom where the receiver can immediately respond without accessing the phone. However, this version might be too intrusive and only work well in quiet and private environments.

3.4.4 Counting People

This service counts the number of persons that are inside a given area. It allows proactively informing the administrative user about persons entering or leaving the geofence. For presence detection the server requires appropriate sensors. It is thus not necessary for people being detected to have the service installed. For example, authorized airport staff could this way be informed about a growing queue at the check-in counters, allowing to take countermeasures.

3.4.5 Switch Service

This service is used to remotely control devices, like lights or shutters. It can be used when the hardware switch is out of range or for remotely switching groups of devices. An enhancement of this service for light devices makes the light follow a user who is moving through the building.

Different users may have different access permissions: For example in a private house hold, children can only control light of room being currently in, parents can control all lights from anywhere within the house, neighbor can only switch off lights and only if no family member is present and no automation has been enabled.

3.4.6 Information Request

Telling this service where you are, will make it provide you with location-linked information regarding a specific question. Services of this type could answer questions, like "Who's office is this?" or "Where is the closest color printer?".

3.4.7 Data Archive

A company can offer its employees a service offering a digital data archive. Users are allowed to access and store internal documents only when being in office. Executives and managers may be allowed remote access via a secured, tunneled connection into the local network.

3.4.8 User Configuration

When approaching a predefined geofence (possibly at a certain time of day or during a specified weather condition), this service activates a setting or a function. Examples include: switching on the light when entering a room or calling the elevator when approaching the office building. In case of multi-user access of hardware devices priorities need to be managed.

This service can also be used for offering vouchers, e.g., used by a restaurant. Provided that this service is installed, whenever a user passes by the establishment a voucher of limited duration is sent.

3.4.9 Home Monitor

The home monitor service checks for and warns about potentially dangerous or undesirable situations. For example, when the last resident leaves the house the service checks whether all doors and windows are locked; when leaving the kitchen but stove is still on; or when the water in the swimming pool needs attention. For detecting according situations the server requires appropriate sensors. Users that were outside while an alarm occurred are immediately notified as soon as the home monitor server is reachable. For very important notifications a secured Internet communication may be used.

3.4.10 Bulletin Board

Users of this service can post and view multimedia messages at virtual, room-linked bulletin boards. When entering a room with new messages, the user is proactively informed. Optionally, messages are stored locally for later reading.

3.4.11 Trace Users

Mobile users signed up for this service, regularly or when changing position, send their own position. This way movement profiles of registered users are created. Authorized users can query for current locations of users which is, e.g., visualized using the navigation service.

3.4.12 Tour Guide

In a museum the tour guide service shows the visitors proactively information about the closest exhibits. The sensors of the smartphone, e.g. compass and microphone, can be used to sort the list by relevance. Additionally, the user can choose an exhibit from a list. The tour guide then activates the navigation service to show a route to the destination.

Further, the user can be informed about upcoming events, e.g. a guided tour or feeding time in a zoo.

3.4.13 Shop and Product Finder

Inside a mall customers can use this service to find shops which offer a certain product. A list of categories as well as a text-based search may be offered. Once the user has selected the entry of interest, the navigation service is started to guide the user.

3.4.14 Reception Service

This service must be made available to the user by the inviting host. First, the host creates an appointment containing room and time of the meeting. This information together with a the navigation service is encoded into a link or QR code and sent to the guest (e.g., via e-mail). When the guest, who is the user of the service, opens the link or scans the code, the service is installed and stays idle in the background. Only when the user enters the building at the time of the appointment, the service becomes active.

It welcomes the guest and starts the navigation service which leads the user to the location of the appointment. Additionally, the host is informed about the arrival of the guest.

3.4.15 Smartphone Reminder

When leaving the office, room, or house – i.e. when the door sensor detects that door is being closed – this service checks whether the owner's smartphone is near by. If it is close but not moving, it was probably left inside the office, and informs the owner audibly about this potential oversight.

3.4.16 Intruder Alert

Having a sensor attached to a door which detects the opening of the door, this service determines whether any of the owners (of the office, house, etc.) is close. If not, an intruder might have opened the door and all owners are being alarmed via their smartphones. Additionally, a general alarm may be raised.

3.5 Differences to Existing Technologies

While LLSs are capable to realize all above described use-cases other, existing technologies may be used as well. This section emphasis the peculiarities of LLSs by comparing them to conventional apps, location-based services, and smart home applications.

3.5.1 LLSs vs. Conventional Apps

Conventional apps and LLSs differ mainly in a single aspect: LLSs are distributed applications by definition and may communicate only with its server counterpart by default. Conventional apps on the other hand may communicate with any Internet server if the according permission is owned by the app which is the case for more than 66% of analyzed apps according to research [FGW11] and [SLG+12].

3.5.2 LLSs vs. Location-Based Services

Even though the name of location-linked services might be misinterpreted as location-based services (LBS), both are not equivalent. According to

section 2.2.1 LBS make use of some type of geographical location. LLS, on the other hand, while being deployed inside a building are not necessarily aware of any location at all, as for example the data archive service from section 3.4.7. However, most LLSs are aware of the location of at least one entity and can hence be considered LBSs. In particular, if LLSs need to limit their detection area within the WiFi covered area, the position of the user is required.

On the other hand, there are location-based services which are not LLSs. E.g. a conventional app that is aware of the user's location by accesses the GPS module is a location-based service. However, as it is not a distributed application it cannot be a LLS. Thus, there exists an overlap between both kinds of services but none is a superset of the other, as visualized in figure 3.2.

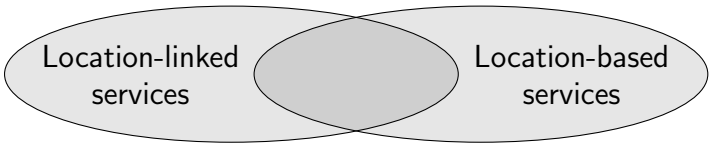


Figure 3.2: Relation of location-linked services to location-based services

3.5.3 LLSs vs. Smart Home Applications

Currently smart home systems provide home automation, intrusion detection, video surveillance, fire detection functions, occasionally even patient health monitoring and entertainment support. They allow owners to automate procedures in their house, react to environmental changes or user-triggered events, and interact with other systems or humans. They are called smart home applications in the following. Unfortunately, there exist a number of problems and unsolved design issues of current smart home systems. Further, there is room for improvement.

LLSs extend and enhance the concept of smart home systems and their applications. While smart home applications are executed on the server, the user's smartphone is merely used for displaying results, usually in a standardized way, and basic user interaction. On the other side, a LLS can be a distributed system, being executed both on the server as well on the

client. The client can thus execute program logic, store data, and show customized GUIs. Reducing the client to a simple browser application, a LLS becomes an ordinary smart home application. A graphical representation of the relation between LLSs and smart home applications including sub-branches is provided in figure 3.3.

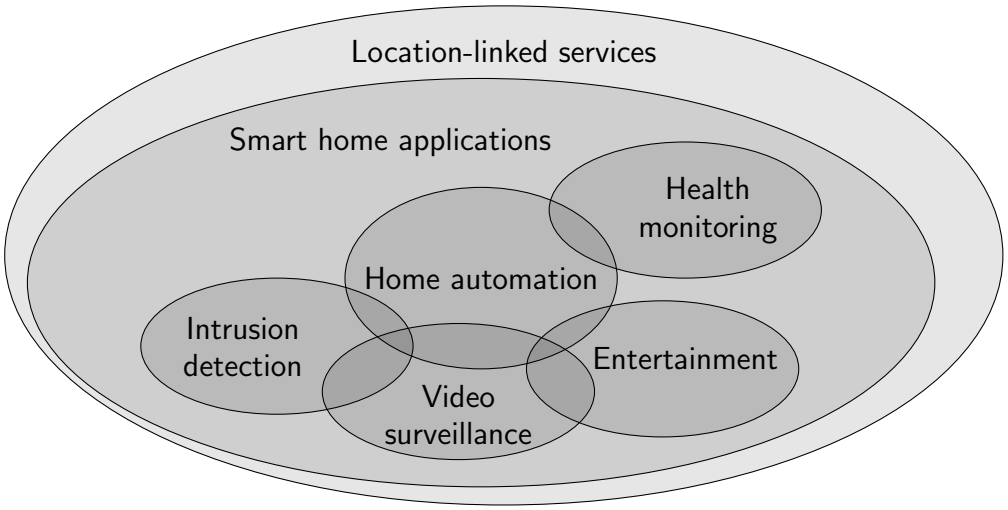


Figure 3.3: Relation of smart home applications to location-linked services

Out-sourcing program logic to the client-side enables services to locally collect and process data. This improves scalability and allows to increase privacy as the server does not need to be an omniscient entity. Further, LLSs are executed as full-value programs on the server side. Consequently, they can act as controller software for smart home systems (cf. section 2.3.2) and thus integrate devices of different vendors.

3.6 Areas of Application for LLSs

LLSs are particularly suited for providing location-linked, sensitive, and private functionalities as introduced in section 2.1.5 due to their design. However, due to the limitation of local access only, LLS are not a replacement for all conventional apps. However, there are a number of fields of application where this restriction does not affect usability or where security concerns outweigh loss of user-friendliness. With this in view, following places and environments are particularly suited for applying LLS to:

3.6.1 Private Homes

Within private homes LLSs are well suited for offering home automation services. For many home automation applications no remote access is required, examples include the automated door bell (3.4.2) and remote switches (3.4.5), personal user configurations (3.4.8) and the home monitor service (3.4.9), the smartphone reminder (3.4.15) and the intruder alert (3.4.16).

Using LLSs in private homes allows a high degree of flexibility and customization and thus allowing the home owners to stay in control and to defined there own personalized GUIs. However, most important is the property of providing a closed system within the local network offering a good measure of privacy and security.

3.6.2 Office Environments

The same property is very desirable for applying LLS to office environments. It allows, for example, to provide navigation functionalities to employees and guests (3.4.1) without making building maps globally available. Similarly the data archive service (3.4.7) can be offered without disclosing data to outsiders and even without making it accessible from the Internet.

Offering a reception service (3.4.14) for guests may increase user-friendliness without requiring the user to manually search and install an app from an app store. Instead invited guests are welcomed proactively.

3.6.3 Public places

A third environment in which LLSs are also well suited to offer functionalities to users are public places like shopping mall, hospitals, airports, and stadiums. In these environments all services are usually usable without authentication. The advantageous strength of LLSs in this case is thus neither privacy nor security, but rather the automatic detection of services. Customers can use indoor navigation services (3.4.1) or product finders (3.4.13) without manually searching and installing appropriate apps for the current building.

In special public places such as cruisers and airplanes, Internet access

may be limited or very costly. Here, LLSs allow an alternative method to provide user services.

3.6.4 Incentives for Using LLSs

Summarizing, four incentives have been identified for using LLSs:

- Security and privacy: All data stays in-house
- User-friendliness: Automatic discovery of services
- Customization of services including personalized GUIs and staying in control
- Offering distributed services even if no Internet available

Chapter 4

Basic Considerations and Requirements

After chapter 3 provided a definition, general description, concrete examples of LLSs and areas of application, the following section presents general considerations about design, usability, as well as user acceptance of a provisioning system for LLSs. In sections 4.2 and 4.3 non-functional and functional requirements are formulated, respectively.

4.1 Design Considerations

The most fundamental requirement for a LLS provisioning system which aims to be accepted by users, is to provide robust and beneficial services while being comfortable and intuitive to use. Further, users must be certain that personal data shared with LLSs is safe.

As explained in sections 5.1.1 and 5.5.3, the provisioning system must comprise an app which is very powerful. It is likely that this app needs to own even more permissions than the already very too powerful deemed Facebook and WhatsApp Messenger apps mentioned in chapter 1. To certify that the app does not use its permissions mischievously or even illegally, this dissertation proposes to make its source code public. This way independent experts can attest the legitimate use of all permissions to help gaining the trust of end users.

In order to provide beneficial services for users, LLSs must be at least

as powerful as conventional apps. By definition (cf section 3.1), LLSs are distributed applications which allow centralized storage and processing of user data as well as user interaction.

In order to prevent aggregation of very much sensitive, personal data, each LLS should work independently and require as little data as possible to be able to perform its tasks. That is, there must be no omniscient entities which collects large amounts of sensitive user data and may potentially deduce even more information. As an additional security barrier all data must stay in-house, that is, where the LLS is offered. Connections to external Internet servers must be avoided. At the same time by keeping LLSs independent of each other, no single point of failure is induced.

In order for any system in general – and a provisioning system for LLSs in specific – to be successful also its development, deployment, and maintenance must be comfortable and intuitive. For this reason configurable systems are often designed in such a way that an average user is supposed to set it up on its own. As a result the configuration must be kept simple and intuitive. Resulting in limited settings options – often set up using a graphical interface or a very simplified programming language. Typical projects of this kind are openHAB or IntuiSec, a framework for intuitive user interaction with smart home security using mobile device [KS07].

However, home automation systems and also location-linked services become quite complex fairly easily. For example, consider a shutter control comprising presence detection of the residents in the living room as well as an outdoor light sensor: Usually the shutters should rise earlier on weekdays than on weekends. However, if on a weekday some residents are sleeping late shutters should go up later. If all residents are awake early, shutters should also rise early, however not before it is light outside. Further, there should be an option to overwrite the automatic behavior. Simplified configuration menus or programming languages cannot model wanted behavior or configuring them becomes disproportionately complex. In either case, it is not manageable by an average user. Thus, a programmer is needed to set up and configure the system. For this reason, existing and known technologies must be applied which allow programmers to develop new LLSs without lengthy initial training.

Finally, in order to easily reach a large audience, LLSs should be platform independent.

4.2 Non-Functional requirements

From previous design considerations a number of non-functional requirements are deduced in the following. According to Borque et all, non-functional requirements constrain the solution. They are sometimes also known as constraints or quality requirements [BF+14, p.1|3]. For later reference worked out non-functional requirements are consecutively numbered preceded by a capital N, e.g. {N2}, and summarized in table 4.1.

#	Summarized description
{N1}	no single point of failure
{N2}	no omniscient entity
{N3}	building provides its own data
{N4}	local communication only
{N5}	authenticated and encrypted communication
{N6}	services as powerful as conventional apps
{N7}	ease of development and administration
{N8}	platform independent services
{N9}	ease of usage

Table 4.1: Summary of non-functional requirements

4.2.1 Robustness

The system must be fault tolerant. When a single service fails, the rest of the system must not be affected. For this reason a distributed system design is to be favored. Wherever possible services should not depend on each other. Ideally each service is offered by a different server device. For example, the door bell service from section 3.4.2 can be offered by a micro-controller embedded in the bell.

No single component shall be introduced into the system which allows a complete breakdown, i.e., any single point of failure must be avoided {N1}.

4.2.2 Omniscient entities

In order to protect user privacy and the private resources, there shall be no omniscient entities {N2}. In particular, no system component must collect, receive, or calculate user locations. From this requirement follows that each mobile device must support offline localization, e.g. must be able to determine its position on its own. There can be no infrastructure which performs the actual localization, as this component would be potentially aware of the locations of all users at all times.

For offline localization mobile devices need a localization database. Further, map data may be required by some services. This kind of building-linked data must be provided by the building itself (or by some entity inside it) {N3}. In particular, there may be no global server managing and offering building-data for multiple buildings. Building data belongs to the building. It must be in direct control of the owner.

For security and simplicity reasons as well as to ensure fault tolerance, user management, authorization, and access control have to be performed by the services themselves. There shall be no login nor authorization server which determines which users may access which services and perform which actions. Such an authorization server would imply global knowledge about known users and available service actions. It would further require configuration when adding a new or modifying an existing service which contrasts the requirement of *Ease of Use* from section 4.2.5.

4.2.3 Direct, Secure Communication

On the one hand, LLSs can control security relevant objects, such as shutters and garage doors. On the other hand, [CBR03] pointed out: Buggy host software is a major security issue; the only solution to run it nonetheless, is to isolate it behind a firewall. As consequence, the provisioning system must not rely on Internet connectivity by default but rather use a local network {N4}. This implies that push message mechanisms provided by mobile operating systems cannot be used as they rely on a third-party Internet server.

If the local network is separated from the Internet, the system cannot

be attacked from outside which reduces the number of potential attackers considerably. In order to protect the system also from malign user within the local network, communication partners need to be authenticated and an encrypted connection must be used – ensuring that the conversation cannot be overheard and inadvertently revealed to third parties {N5}.

If remote access is inevitable by a service, a secured, tunneled connection into the local network must be established first. However, this is not in the scope of this dissertation.

4.2.4 Flexible and Multi-Platform Development

As stated in section 3.3, a service should be comparable powerful as a native app {N6}. Existing technologies and programming languages should be used where appropriate. Learning a new language in order to create a LLS is inadequate. Instead the framework envisioned by this dissertation should make developing new and administering existent services as easy as possible {N7}. At the same time both the server as well as the client part of a service shall be platform independent to keep developments efforts low {N8}.

4.2.5 Ease of Use

A major demand to a provisioning system is ease of use {N9}. A common smartphone user must be able to use the provisioning system, including installing and setting up the client side as well as launching and using services in any building.

Also for administrators the provisioning system should be easily manageable. New services must be able to be installed without changing any system wide configuration settings which would violate requirements of no single point of failure {N1} and also the requirement of no omniscient entity {N2}. Instead installing a service should be as easy as starting a program on a network attached device.

4.3 Functional Requirements

As formulated by Bourque et al.: Functional requirements describe the functions that the software is to execute; for example, formatting some text or modulating a signal. They are sometimes known as capabilities or features. A functional requirement can also be described as one for which a finite set of test steps can be written to validate its behavior. [BF+14, p.1|3]

For this dissertation two different sets for functional requirements are to be considered. On the one hand, the provisioning system needs to fulfill a set of demands, on the other, each LLS needs to be able to execute a set of functions.

Following functional requirements are derived from the detailed description of a LLSs provisioning system (Sec. 3.3) as well as the example service from section 3.4. A summary of all function requirements is shown in table 4.2.

4.3.1 Functional Requirements of a LLSs System

In the following subsections the functional requirements are listed which must be fulfilled by a LLSs.

Service Detection and Filtering

The system must support an auto-detection mechanism allowing all users (including guests) to find all entities offered by the provisioning system {F1}.

Next, found services must be filtered. Too restrictive filtering will hide the service from users, while too lax filtering will clutter the list of available services. A system being able to provide filtered detection of all services from section 3.4, must filter by at least two filter criteria: location and user {F2}. Specific values for these filter criteria for the sample use cases are given in table 4.3. Other criteria like time, weather, or context are not considered at this point.

After filtering LLSs, the system must notify the user about available services, e.g. by presenting a list {F3}. Therefore, each service must have a

#	Summarized description
{F1}	find services
{F2}	filter services
{F3}	list services
{F4}	show service description
{F5}	trust level for services
{F6}	services must authenticate themselves
{F7}	users must authenticate themselves
{F8}	services requires permissions for security-related functions
{F9}	inform user about permissions being used
{F10}	localize user
{F11}	support offline services
{F12}	two way communication
{F13}	service update mechanism
{F14}	user can launch service manually
{F15}	wake event: other service can invoke service
{F16}	wake event: by geofence
{F17}	wake event: on receiving server push message
{F18}	wake event: on timer
{F19}	wake event: when service provider becomes available
{F20}	services can be running in background
{F21}	client part: flexible GUI
{F22}	client part: play audio
{F23}	client part: user notification
{F24}	client part: use accelerometer
{F25}	client part: use compass
{F26}	client part: access building map data
{F27}	client part: access user position
{F28}	client part: initiate server communication
{F30}	client part: invoker other service
{F31}	client part: use private storage
{F29}	client part: receive server push messages

Table 4.2: Summary of functional requirements

Service	Detection area	Allowed users
Navigation service	whole building	anybody
Automated door bell	near entrance	authorized guests
Audio messenger	whole building	authorized users
Counting People	administrator's office	administrator
Switch service	default: room, janitor: building	anybody (janitor is authorized user)
Information Request	whole building	anybody
User Configuration	in the proximity of controlled devices, usually: room	owners
Home Monitor	whole building	owners
Bulletin Board	whole building	anybody
Trace Users	administrator office	administrators
Tour Guide	whole building	anybody
Reception Service	via link only	link receiver
Smartphone Reminder	office	owners
Intruder Alert	office	owners

Table 4.3: Values for location filter and user filter for example services

name and provide a description {F4}. Further, the system must assign trust levels for each service {F5}. They may warn users about potentially malicious services or services offered by untrusted entities. By default only trusted services should be installable. However, users must be able to override values determined by the system.

Authentication, Authorization, and Permissions

Before installing a service the user must know whether it is trust worthy. Thus, services have to authenticate themselves to users {F6}. Based on the provided authentication the user can decide whether to install (and hence

trust) the service or not.

Also users have to authenticate themselves if asked for by a service. User authentication must be offered by the system {F7}. Services can rely on the user authentication obtained from the system. However, authorization is left to the services. They may provide different quality of service to different users or reject access all together.

Even when connected to a trusted service, the user needs to be aware of private or personal data which is shared with the service. The user must be able to inspect and restrict service permissions at all times {F8}. Further, the user should be proactively informed when a service actually makes use of its owned permissions {F9}.

Localization

To be able to provide location-based services and to provide filtering by user location, the system needs to be able to determine the geographical position of users {F10}. Considering the use cases from section 3.4, primarily a localization inside the WiFi network is needed of room-based granularity. More precise or outdoor localization is optional. In order to avoid an omniscient entity which is aware of the locations of all users ({N2}) , localization must be performed by the mobile devices and only when a user position is required. It must not be performed by a localization infrastructure.

Service Launch, Communication, and Update

While for a distributed system a network connection is essential, a LLSs system must still be able to operate – with limited functionalities – in offline mode {F11}. This implies that LLSs can even be launched when their corresponding servers are not available. Missing server connectivity must be detectable by LLSs.

Whenever the corresponding server is available, the LLSs system must provide a two-way communication {F12}. This implies in particular, that a server may push messages at any time to its service clients. This must even be possible, if the corresponding client is not running. In this case

the message must be cached and delivered as soon as the client becomes available again.

In order to extend functions and capabilities, the LLSs system must provide means to update services {F13}.

4.3.2 Functional Requirements of LLSs

As mentioned above, LLSs need to be comparable powerful as conventional apps. In particular this means that LLSs need a flexible GUI, be able to run as background process, access system and framework functions, and to interact with each other. These four properties are detailed in the following sections.

Activating LLSs

Usually a LLS is started manually by the user. However, conventional apps can also be started when system events occur (cf. section 2.1.2). Considering the example services described in section 3.4, a LLSs system needs to support following wake events:

- user launches service {F14}
- service is invoked by other service {F15}
- user location triggers geofence event {F16}
- push message is received {F17}
- a previously scheduled alarm is fired {F18}
- the service's server becomes available {F19}

In general, whenever one of these events occurs, the LLSs system must inform all services which subscribed to the event. If the service is not active, it needs to be started. Except the first two wake events, all other events should only be active when enabled previously by the LLS. This is necessary to avoid unwanted and repeated launches of LLSs, e.g. every time their corresponding server becomes available. In other terms, the latter events

should activate a LLS only if it was already running. As multiple services need to be able to run at the same time, LLSs running in background need to be supported {F20}.

Table 4.4 lists which wake events are used by each example service.

Service	user launch	invoked by service	geofence	push message	scheduled alarm	server available
Navigation service	✓	✓				
Automated door bell	✓		✓			
Audio messenger	✓			✓		
Counting People	✓			✓		
Switch service	✓		✓			
Information Request	✓					
User Configuration	✓		✓			✓
Home Monitor	✓		✓	✓		✓
Bulletin Board	✓		✓	✓		
Trace Users	✓		✓		✓	
Tour Guide	✓		✓		✓	
Reception Service	✓		✓			
Smartphone Reminder	✓			✓		
Intruder Alert	✓		✓			

Table 4.4: Events by which each example service can be woken

User Interface

LLSs must have a flexible GUI {F21}. Standard elements like labels, text fields, input forms, buttons, radio buttons, canvases, and multimedia containers need to be supported. Further, LLSs should be able to customize

the rendering of those elements. The goal is to put no constraints on the way LLS and user can interact.

Also audible signals need to be supported {F22}. This comprises playing audio files while the LLS is running as background process.

Finally, LLSs must be able to issue system user notifications {F23}. These are usually displayed in the system's header bar and consist of text and images. Additionally, when activated a notification sound is played which obeys the system's notification volume.

Calling functions provided by MOS and LLSs system

As services are similar to smartphone applications and can thus access and forward private data, a permission system is required which limits access right and capabilities of services. This system should be similar to the permission system provided by MOSs (cf. section 2.1.4). The less permissions a service earns, the less harm can it do in case it should turn out malicious. E.g., a service which cannot access localization data, cannot disclose the user's positions; a service without network access cannot transfer any data.

While the number of system functions offered by MOSs is very large, the example services from section 3.4 only make use of following three:

- issue user notifications {F23}
- use acceleration sensor {F24}
- use compass {F25}

Next to system functions provided by the MOS, also the LLSs system needs to provide functions which can be used by services which own the according permission. The example services from section 3.4 require following functions. They thus need to be supported by the LLSs system. For each function must exist an according permission:

- access map and navigation data {F26}
- access user position with room granularity {F27}
- communication: client initiated server communication {F28}

- communication: server to client push messages {F29}
- invoke other service (and share data) {F30}
- private file storage {F31}

While the number of functions – and thus permissions – used by the example services is low, there exist many more conceivable functions which are useful for other LLSs. E.g., if a service should be able to initiate a telephone call, send an SMS, and read contacts from the local address book, according permissions would be necessary.

Table 4.5 details which example service requires which functions and must hence according permissions.

Service	Access map and navigation data	Access user position	Client to server communication	Issue user notification	Invoke other service	Private file storage	Server to client push messages	Use acceleration sensor	Use compass
Navigation service	✓	✓							✓
Automated door bell		✓	✓						
Audio messenger			✓	✓			✓		
Counting People			✓				✓		
Switch service		✓	✓						
Information Request		✓	✓						
Data archive		✓				✓			
User Configuration		✓	✓						
Home Monitor		✓	✓				✓		
Bulletin Board		✓	✓			✓	✓		
Trace Users		✓	✓		✓				
Tour Guide		✓			✓	✓			✓
Shop&product finder			✓		✓				
Reception Service		✓	✓		✓				
Smartphone Reminder		✓	✓	✓				✓	
Intruder Alert		✓	✓	✓					

Table 4.5: Required permissions (sorted alphabetically) for each example service (sorted in order of occurrence)

Chapter 5

Design of Service Provisioning System

This dissertation is at the crossroad of many research areas, including: Smartphone software, distributed systems, ambient intelligence, indoor localization, multimedia and building maps, service discovery, and security. To be able to present a prototype of a service provisioning system which combines all these research areas in chapter 6, this chapter provides an overview about major design choices necessary in order to fulfill the requirements from the previous chapter. The here developed design was the result of theoretical considerations and practical testing.

5.1 System Overview

The system architecture is strongly influenced by technical aspects and requirements presented in chapter 4. An overview of the resulting system structure is summarized in this section.

5.1.1 System Architecture

Since the only officially supported method to add new functionalities to a smartphone, is to install apps (cf. Sec. 2.1.2), this dissertation also requires an installed app. It is assumed to be pre-installed. It is further assumed that the client is connected to the local network via WiFi. Us-

ing this connection the app performs all responsibilities of the provisioning system on the smartphone, including installing, running, and managing new services as well as displaying their graphical user interfaces. As an important responsibility is finding and loading services which are comparable to conventional *apps* and since there are usually *multiple* services, this app will be called *MultiApp*. The complete system will be called BLESS, standing for *Building-Linked, Expeditious Services System*.

BLESS services are distributed applications. They are offered by a service provider, which can be a powerful server belonging to a building or a micro-controller embedded in some component of the building, e.g. the elevator. The service provider executes the server part (SP) of the service. The number of hosted services is only limited by hardware performance. The SP announces its presence and offers the client part (CP) for download. One SP can communicate with many clients, i.e. instances of MultiApp. If external devices (except the user's smartphone) are to be controlled by a service, they must be connected to the service provider; i.e., they must be controllable by the SP. For achieving platform-independence for SPs (requirement {N8}), Java will be used in this dissertation. However, no technical dependencies are introduced, so that instead of Java also the QT library for C++ could be applied.

The CP can be queried for and be installed by MultiApp. Considering that a service must be able to interact with the user while the service provider is not reachable, it must contain program code which has to be executed on the client.

Further, BLESS introduces the concept of buildings and sub-buildings to fulfill requirement {N3} and to avoid omniscient servers storing building data of multiple buildings (requirements {N1} and {N2}). They are like services discoverable network entities and provide data – including map and navigation data – about the areas in which services are hosted. Details are presented in section 5.6.

Another discoverable network entity is the BLESS pusher. It is introduced in section 5.10 and an essential part of BLESS's push messaging system required for fulfilling requirement {F29}.

Previously introduced names of BLESS entities are summarized in the fol-

lowing section. Figure 5.1 visualizes the interaction between BLESS service and MultiApp.

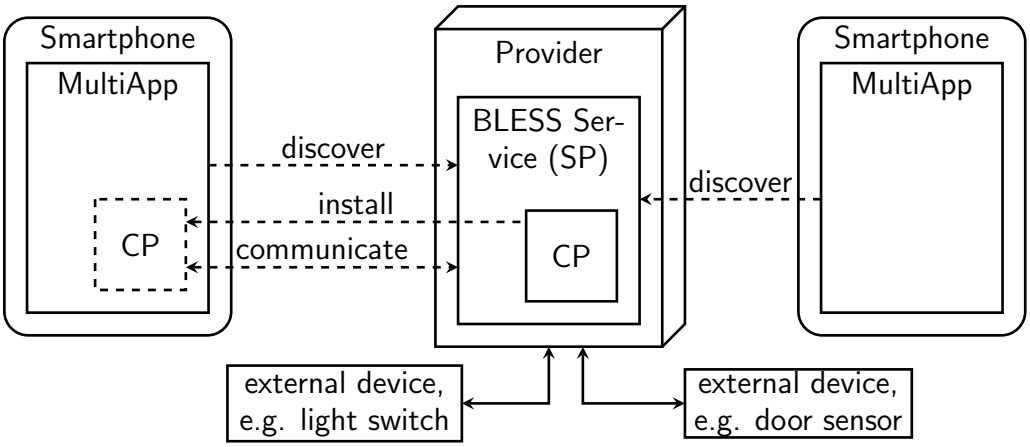


Figure 5.1: Overview of interaction between BLESS service and MultiApp

5.1.2 Naming Conventions

Following list summarizes the naming conventions for integral parts and the provisioning system itself.

BLESS *Building-Linked, Expeditious Services System*, the services provisioning system presented by this dissertation.

MultiApp Smartphone application which is the entry point for BLESS on the client side.

Entity BLESS software components which are discoverable by and interact with MultiApp, these are: The server part of services, buildings and sub-buildings, as well as pushers.

Provider A computer system or micro-controller hosting a BLESS entity.

Service An entity providing a BLESS service which consists of a server part (SP) and a client part (CP).

Server Part (SP) The runnable program which offers the service functionalities. It is bundled with the CP.

Client Part (CP) Executable code which is transferred from the SP to the client and runs inside MultiApp.

External devices Hardware components connected to a provider and controllable by an SP.

Building and sub-building Entities providing data about the area in which services are hosted.

Pusher An entity responsible for delivering push messages.

5.2 Mode of Communication

A pessimistic but realistic assumption is made by Cheswick et al. which can be summarized as: All network programs are buggy. As consequence all network programs can be hacked [CBR03].

Considering that location-linked services are meant to be used in a fixed physical area only, there is no need to access them globally via Internet. It thus seems reasonable to limit access to the system to that area. Hence, service providers are not connection to the Internet. If this restriction is enforced strictly, any potential attacker must be physical present. This limits the number of attackers drastically and thus provides an essential contribution towards increasing security. For these reasons BLESS will exclusively use WiFi for communication between SP and CP, fulling requirement {N4}, and assumes that the local network is strictly separated from the Internet.

5.3 Multi-Platform Approach

Quoting from [1]:

> The diverse and continually evolving MOS landscape constitutes a huge challenge for application developers. Unlike the desktop computer market, where more than 90 percent of users use Windows, mobile app developers must target multiple platforms to reach the same number of users. However, MOSs come with their own software development kit (SDK), each of which

> uses a unique programming language [CL11], and provide their own custom API. In short, developers must write an application separately for each mobile platform. Mobile app providers thus face a dilemma: either invest considerable resources developing the same app for all mobile platforms, or leave some platforms unsupported and risk alienating potential customers. [...]
>
> Anthony Wasserman outlined two options for reducing application development efforts while still supporting multiple mobile platforms [Was10]: use Web browsers to create platform-independent apps, or use cross-platform mobile development tools (XMTs) to create apps for different smartphone platforms from the same code base.

As the CP of a BLESS service is comparable to an app, same challenges arise here. It thus seems a reasonable approach to adopt technology for cross-platform mobile development for BLESS services. Since pure browser applications are neither able to access the integrated smartphone compass nor able to discover network services, only XMTs are considered in the following as valid option.

All XMTs presented in cited research create apps independently executable from any external interpreter or virtual machine. This is not required nor wanted for BLESS. Instead the CP should only contain program code directly belonging to the BLESS service. Any additional software components which are required for all services, should be shared and thus be included within MultiApp. As consequence, only an open-source XMT may be used which can be adapted as required. Of currently available tools from the above cited paper, this leaves Apache Cordova[1], MoSync[2], and RhoStudio[3]. Applications created with RhoStudio require about 3 seconds to launch and have a file size of at least 2 MB. The used virtual machine is thus quiet heavy and is for this reason discarded.

The reason for favoring Cordova over MoSync, is MoSync's complex compilation process. Depending on the target platform, apps can contain

[1] https://cordova.apache.org/ (formerly known as PhoneGap)
[2] https://github.com/MoSync/MoSync
[3] https://github.com/rhomobile/rhostudio

native code or MoSync byte code. In case of byte code, the in the app included virtual machine (VM) can either directly interpret the code or integrate an ahead-of-time compiler that compiles MoSync bytecode to native code on the smartphone. Even though the technical details are well documented, integrating MoSync's VM inside MultiApp and creating a compile chain for translating BLESS services into MoSync byte code, without impacting performance is a highly complex task, which is deemed to be unnecessary for this dissertation. Further, MoSync creates different application packages for different target platforms. The SP would thus have to contain multiple CPs and serve the correct file for each requesting MultiApp instance. This again introduces unnecessary complexity both at compile-time as well as at run-time. It further infringes to some degree the requirement of platform-independent services {N8}. Finally, MoSync is no longer being maintained.

For completeness reasons, a newer approach called *WebAppBooster* introduced by Puder et al. [PTM14] is to be mentioned. It is a technology allowing Internet pages to access local system resources which are beyond HTML5 and JavaScript in a secure manner. WebAppBooster is a mobile app offering an embedded web server, which authorized web sites can use via WebSockets to access system functionalities. Even though it is intended for Internet applications, it can also be used to view and execute local websites. It is thus comparable to Cordova. However, it promises better performance because the web application is executed in the default browser instead of an embedded WebView. On most platforms the WebView browser widget is not optimized in the same way the standalone browser app is [PTM14]. Only with version 7 released in Q3 2016, Android starts to ship only a single browser engine which is updatable – hence optimized – and used by both browser and WebView element.

Even though the source code of WebAppBooster is available, integrating it into MultiApp is expected to be much more costly than Cordova because it is the outcome of a single research project. Further, the libraries offered by Cordova are much more powerful than the functionalities integrated into WebAppBooster. Finally, with Android 7 no performance gain is to be expected anymore.

5.4 Services

Summarizing previous section, BLESS will use the Cordova approach for realizing platform-independent CPs of services. A CP application will thus be a kind of web application. Its logic is programmed in JavaScript.

5.4.1 Installation and Functions

On installation, the CP of the services is transferred from the SP to MultiApp and persistently stored. Here it can be started even if there is no connectivity to the SP, fulfilling offline mode requirement {F11}. MultiApp interprets the code inside a browser element which is also used for displaying the GUI. This allows creating highly flexible GUIs fulfilling requirement {F21}. MultiApp further exposes any required system functionality to the JavaScript code, so it is accessible by the CP of services making the CP as powerful as a native app and fulfilling requirement {N6}. In particular all required functions accessible by CP can be implemented this way. This approach further fulfills requirement of facilitating development {N7} by using existent technologies and the requirement of platform independence {N8}.

5.4.2 Background Services

The browser element in which the CP of a service is executed requires system resources. The more services are running – in foreground or background – the more resources are used. Further, apps running in background can usually be terminated by a mobile operating system if resources need to be freed. Thus, keeping multiple services running within browser elements is to be avoided. It is not required, either, as services are always displayed full screen and thus only one service can be in foreground at a time.

Instead MultiApp closes services which are sent into background. This implies that services cannot perform background operations, e.g. lengthy calculations. However considering the example services from section 3.4, there is no need for background operations. They will thus not be supported. Instead wake events will be offered.

Three different running states for services are distinguished: *stopped*,

running, and *minimized*. The state *stopped* means that the service was not started, yet, or it was explicitly closed. It can only be manually started by the user by selecting it from the services list introduced in section 3.3. A service is *running* while it is being executed and being displayed to the user. In *running* mode a service can register for one or more wake events. If it is closed then, its state changes to *minimized*. It thus does not consume resources as its browser element was destroyed but it can be activated by MultiApp on wake events. The service thus appears to be running in background, on the one hand fulfilling requirement {F20}, on the other not consuming additional resources.

Wake events which need to be supported are listed in table 4.4. They correspond to requirements {F14} through {F19}. The wake events *user launch* and *launch by other service* are special, as they are always enabled. All others need to be activated by the CP during *running* state. This is done by JavaScript methods offered by MultiApp.

While a service is registered for some wake event, MultiApp itself listens for the corresponding event. Once it occurs it ensures that the registered service is activated and informed about the occurrence of the event. If MultiApp is closed while services are *minimized*, they cannot be wakened. It is thus assumed that MultiApp is running at all times.

5.5 Security

One of the most important attributes for a services provisioning system which aims to handle private data and to control security relevant devices, is security. Users must be certain that their data is only transferred to trusted entities and that services only accept control commands from authorized users. A bilateral authentication is thus required as well as encryption for network traffic.

However, a framework for LLSs cannot secure services by itself. It can rather enforce security policies and provide mechanisms which can be applied and used by services and users to increase security. The first essential contribution in increasing security is the design choice of relying on local communication only as introduced in section 5.2. Further contributions are

the introduction of certificates used for authentication and encryption as well as permissions for limiting capabilities of CPs. These are presented in the following subsections.

Computer and network security is a highly complex topic. Full security can never be guaranteed. A completely safe system can thus not be the goal of this dissertation. For this reason no details about how to use certificates for authentication and encryption are presented. The intention is rather, to lay out the foundation for a state-of-the-art security concepts for the system which can be configured and enhanced after a thorough security audit of the working system at a later time.

5.5.1 Authentication and Trust Levels

For authentication either a ticket system can be used or authentication can rely on certificates. Using the former approach requires an authentication server which knows all services and users and would also present a single point of failure which violates requirements {N2} and {N1}, respectively. The same issue arises when applying the certificates approach and introducing a root authority as PKI.

Consequently, this dissertation relies on certificates without PKI. Specifically OpenPGP was chosen due to its open specification and wide usage. Existing certificates can thus be used if available.

It has to be noted that in order to fulfill requirement of local network only {N4}, access to public key servers has to be circumvented. Details are explained in the following.

Entity Authentication

When MultiApp detects a new entity its trust level needs to be set according to requirement {F5}. A service can either be trusted or untrusted. Further, there is an option when the user needs to decide whether to trust a service. This concept is applied for all entities.

Since BLESS is a framework open for anybody to create new services, it is impossible to predict which service is trustworthy. MultiApp could integrate sophisticated code analysis tools which might be able to determine

what the CP of a service does, however, the code of the SP stays hidden. There are thus no means to determine what happens to data transferred to it. For this reason BLESS does not attempt to interpret code to judge its trustworthiness.

Instead BLESS uses OpenPGP certificates for all entities in order to fulfill requirement {F6}. For each service a dedicated certificate is to be used. Discovery responses are signed using that certificate and it is later used to establish secure communication. For determining whether to trust a certificate, MultiApp relies on OpenPGP's validation procedure as summarized in section 2.5.2.

For private households and small companies in which BLESS entities are developed by a family member or an employee, respectively, it is considered likely that users know the author in person. It is further assumed that users already trust the author's certificate and can make it available to MultiApp. This way, entities fully trusted by the author are also trusted by the end users.

For public facilities and buildings in which BLESS entities are developed by a third party, building and service certificates are proposed. The building certificate is created by the building owner or administrator and is the root of the certificate chain for the building. For each entity a dedicated certificate is created. These are signed as fully trusted using the building certificate. This way users only have to explicitly trust the root certificate and can be certain that only officially supported entities will be used. Responsible handling of the building's root certificate is assumed.

The public part of the building certificate must be made available to users via a secondary, secure channel. As a public key server cannot be used as it would require Internet access, the building certificate could be stored on some server inside the local network. The internal URL is then encoded as QR-code. This code could, e.g., be shown to all guests entering the building inside a locked display case in the entrance hall. MultiApp would need to offer an option to scan that QR-code, download the linked certificate, and installed it after asking the user whether to trust it. For security reasons the QR-code should additionally to the URL also contain a hash value of the certificate. Note that using this approach service can only be used after

entering the building and scanning the code. Using door bell service (3.4.2) when visiting the building for the very first time is thus not possible.

User Authentication

For security relevant services also user authentication is of major importance. Next to using user certificates, also user names and passwords could be used. This later approach, however, is not user-friendly as it requires users to enter – possibly for each service different – login data. Further, as there is no centralized authentication server, changing a compromised or expired password is costly. A certificate is assumed to be changed less often. It can further not be stolen verbally nor using key loggers. Thus, user certificates are used in this dissertation for user authentication in order to fulfill requirement {N7}.

Ideally, a user already owns an OpenPGP certificate which is trusted by friends and other contacts and which is already being used for signing and encrypting e-mails. If this certificate is available on the user's mobile device, it could also be used by MultiApp for authentication. The authors of a BLESS service need to be aware of the users certificate. They further need to make the service trust the certificate.

Otherwise, a new certificate must be created, e.g. by MultiApp itself, and must be made available to the service author in a trusted manner. This could be for example be achieved if the users visit the author in person and prove that they are able to decrypt a message which was previously encrypted by the author using the users certificate. The author can then add the certificate to the service's certificate store and declare it as trusted. In either case, it is assumed that users own a personal certificate and the public key is made available to BLESS.

In a second step appropriate access control rights have to be granted for the certificate owner.

Mutual Authentication

Having certificates of both communication parties in place, mutual authentication becomes possible. At this point the design of BLESS is intentionally

kept open as there is no standard method available. SSL/TLS could be applied enhanced with user authentication or a custom mutual authentication protocol could be introduced. Thus, even though the usage of certificates for all communication parties is envisioned, the actual procedures for authentication (requirement {F6} and requirement {F7}) and hence setting up secure communication channels (requirement {N5}) is not included in this design.

5.5.2 Encryption

All communication except for discovery requests and responses need to be encrypted according to requirement {N5}. This could either be accomplished by encrypting every message using the receiver's public certificate. Alternatively, an SSL connection can be established. Using certificate encryption both communication parties then agree on a shared key which is afterwards used for symmetric encryption.

5.5.3 Service Permissions

At this point users, who are about to install and use a service, can be certain of the author of the service. Further, they can assume that the communication channel is secure. However, there is no way to give any guarantee what the service does with shared data. For this reason service permissions are introduced. The major idea is to limit the amount of data a service can access and is thus able to share.

For this reason MultiApp requires the user to consent before the CP of a service can:

- access the user position (requirement {F27})
- communicate with its SP (requirement {F28})
- receive push messages from its SP (requirement {F29})
- invoke other services (and pass data) (requirement {F30})

The other permissions listed in table 4.5 are not considered security relevant. User consent is thus not necessary. However, it might be added at a later point to allow better user control.

The user is asked for consent before a service is about to use the permission the first time. Options are *grant always*, *grant this time*, *deny always*, and *deny this time*. Additionally, the user can view a list containing all permission choices for a service and change each setting. This fulfills the service permission requirement {F8}.

It has to be noted that MultiApp requires all permissions any LLS could request. This results in a very powerful app which could easily publish a great amount of private user data. As proposed in section 4.1, to gain the users' trust anyway, the source code of MultiApp should be made public.

5.6 Buildings and Service–Buildings–Linkage

As services are linked to buildings, a consistent notion a *building* and *service* needs to be defined. Further, an appropriate service-to-building linkage is developed in this section.

5.6.1 BLESS Buildings

A building in the general sense is structure which can be entered by humans usually delimited by walls and a roof. In this dissertation, however, it is used as synonym for the area covered by a WiFi network. This often corresponds roughly to a physical building. However, in case of large institutions, like universities, hospitals, or companies, a WiFi network may cover a whole building complex, i.e., multiple physical buildings.

In this dissertation *building* may refer to both: a building in the general sense or a campus-like structure. If the context does not clarify which is meant, a building spanning several sub-buildings will be called *building complex* or *campus* from now on.

According to table 4.3 all services detectable in the whole building, are actually linked to the building complex. For example, considering a navigation service 3.4.1 for a university campus or a tour guide 3.4.12 for an open

air museum, all buildings belonging to the same campus must be covered by these services. For simplicity, a service is always linked to a building. Linkage to a sub-building is not supported.

When entering a building, its services are to be presented to the user. At the same time, the user should have the option to list all services which where installed in other buildings. A unique identification for each building is thus needed. Further, a human-readable name is helpful for the user to distinguish buildings easily (requirement {F4}). As providing building identification and name through each service would imply duplicated and thus potentially inconsistent information, an entity representing the building will be introduced fulfilling requirement {N3}.

This building entity needs to be detectable by MultiApp and thus needs to be an auto-discoverable network service. For consistency, the same discovery protocol as BLESS services implement will be used. It is detailed in section 5.9.3. Further, these building services will provide localization data to enable offline localization within the building. They also supply a building map which can be used by all services belonging to the building. For building services assumption 1 must hold:

Assumption 1. *Within each WiFi network there is at maximum one BLESS building service available. If BLESS services are offered there must be exactly one building service running.*

As BLESS buildings may not overlap and with assumption 1 applied, MultiApp can unambiguously determine in which building it is residing, by detecting building services. Further, it can thus determine to which building a service is linked. It is thus not necessary for services to explicitly state its building-linkage which is in line with requirement {N9} (ease of usage and administration). Then the localization data of the building can be used to determine a more accurate position inside the building which in turn is used for the filtering mechanisms described in section 5.9.

5.6.2 BLESS Sub-Buildings

A building containing sub-buildings – that is a campus – needs to represent each sub-building by a dedicated network service. Also these sub-building

services provide a name, unique identification, localization data, and optionally a building map. Additionally, they need to provide data about their dimension and location. Using these information MultiApp is able to determine when a sub-building on a campus is entered. This is necessary to determine a more accurate position inside sub-buildings using the localization data of the corresponding sub-building. Areas of sub-buildings must not overlap. This can be ensured by using real, physical building dimensions.

For example, a university campus providing BLESS services will have a campus service providing a campus map and localization for the campus. Further, there are sub-building services each providing a building map and data for indoor localization of the building it belongs to.

5.6.3 BLESS Services

An important meta design choice pursued by this dissertation is keeping development of BLESS services as easy as possible while offering high flexibility. An important component to keeping complexity low is by avoiding unnecessary interconnections between services, buildings, and sub-buildings. E.g., it should not be necessary to know the exact building identification during development of a service. It should further not be necessary to make a new service known to building or sub-building services. Instead it should be sufficient to know where the service is to be installed, i.e. geo coordinates and if required room numbers. This eases development as demanded by requirement {N7}.

Providing an implicit service-building-linkage is easily possible because there is always exactly one building service available within a BLESS WiFi network. All detectable services belong to that building. Thus, if the name or the identification of a building changes or its services are moved to another building, no further modifications are necessary.

As can be seen from table 4.3 about half of the sample service should be detectable and thus installable not in the whole building but rather a small section of it. This dissertation proposes to use a single room as smallest possible detection area in order to fulfill requirement {F2}. This is a plausible approach as most buildings are divided into rooms and rooms can be

distinguished easily. In case of very large rooms, they may be split into sections with each section being a virtual room. Alternatively, the detection area might be specified using geo coordinates. However, precisely determining latitude, longitude, and altitude inside buildings is a non-trivial task, both for the administrator during design time as well as for the localization algorithms of MultiApp while operating the system. Further, as suggested in section 2.2, it is not to be expected that WiFi localization will yield much better results than room-based localization. Defining detection areas for services inside buildings via geo coordinates will thus not be supported.

As will be described in section 5.7, indoor positions will be provided as discrete room labels by MultiApp's localization algorithms. Services which need to narrow their detection area to a set of rooms are required to use the same room labels. Thus, a linkage between these services and their parent building via room labels is inevitable. To minimize difficulties due to naming and spelling differences, this dissertation proposes to use already existing room numbers as labels where ever possible. Labels containing names of people working or living in a room are to be avoided as they are subject to change.

Even though services are always linked to a building, their detection area can also be limited to a set of rooms in a sub-building. In this case the service is said to belong to the sub-building. For such services an unambiguous mapping is imperative, since identical room labels may be used in multiple sub-buildings. Unfortunately, even though the service-building-linkage can be determined autonomously due to BLESS's structure, in order to link a service to a sub-building, additional input has to be provided by the developer of the service. Again to avoid ambiguities due to naming and spelling differences, this dissertation suggests that each service belonging to a sub-building must provide geo coordinates of its location. The coordinates may be an estimate and must only be located within the borders of the sub-building. Since the dimensions of sub-buildings are known, MultiApp can use the service position to map service to sub-building. Alternatively also the unique identifier of the sub-building may be provided. Note that if no service location is provided a service always belongs to a building rather than a sub-building.

5.7 Localization

Since omniscient entities – including those that are aware of the locations of all users – are to be avoided according to requirement {N2}, MultiApp needs to be able to determine the user position autonomously. Further, no external hardware should be necessary such that the system can be applied to any building without additional infrastructure costs (cf. section 3.3).

According to section 2.2 WiFi localization using fingerprinting seems to be the most promising approach, it is applied here for the inside of buildings in order to fulfill localization requirement {F10}. Locations will thus be given as discrete labels each standing for a room. However, if a localization technique provides continuous positions as geo coordinates, those may be unambiguously translated to a room label using the localization and map data provided by building services. If available, geo coordinates shall be passed from MultiApp to its services.

Also localization outside buildings will be supported. Only hardware integrated into smartphones may be used. Outside locations are specified as geo coordinates using latitude and longitude. The altitude for outdoor positions is not be required by BLESS.

In order to prevent location aware entities, a server-based location query API cannot be used. It would know where its users are by analyzing the queries it receives. Also infrastructure-based or other commercially available systems cannot be applied in this project.

Instead BLESS will include a custom localization framework which allows buildings to make any number of radio maps available to its users. The localization algorithms are integrated into MultiApp. Depending on which kind of radio map is offered by a building the according localization algorithms is used. If multiple radio maps are available, the first algorithm provides a coarse location which is improved by subsequent algorithms. Note that using this approach, any offline localization technology can be supported. Even though WiFi localization is used here, e.g. also a beacon-based approach could also be adopted by storing location data about the individual beacons within the buildings' localization databases.

Every localization procedure costs processing power and thus battery

life. Localization must only be performed when needed.

5.8 Navigation

In order to make use of the position provided by the localization system of BLESS, a map technology is required which fulfills the requirements. Unfortunately both Google Indoor Maps as well as MapsIndoors require an Internet connection and can therefore not be used. The commercial products from section 2.2.5 cannot be used in this project due to their price.

The free project OSM could be used for BLESS. However, creating maps and especially indoor maps for OSM is very complicated and time-consuming. For this reason a simpler map technology was created which is presented in the following.

The remainder of this section 5.8 is an extract from [3].

The presented system - SvgNaviMap - is intended to provide indoor navigation on smartphones. It bases on existing floor plans without modifying them. Thus, colors and outline of maps are the same on wall-mounted maps and the navigation application. This facilitates orientation for the user as recommended by [PSH+09], reduces workload for creating maps, and enables a uniform corporate identity through maps on different media. Further, emphasis is put on easy creation of navigation models. To the best of our the knowledge SvgNaviMap is the first open-source project which allows navigation on existing 2D maps across several floors.

The basic work flow for making existing maps navigable is shown in figure 5.2. First, a floor plan of the building in question in the SVG format is required (a). This map is then loaded with the SvgNaviMap editor. Visually routing information is drawn on top on the map (b). This includes defining points of interest (POIs) and grouping them into categories. Further, the map has to be pinned to a global coordinate system (c). After the configuration phase is completed, the map including navigation information is supplied to a smartphone, e.g., via a web server (d). Assuming that a position provider is installed (e); the user can choose a destination and start navigation.

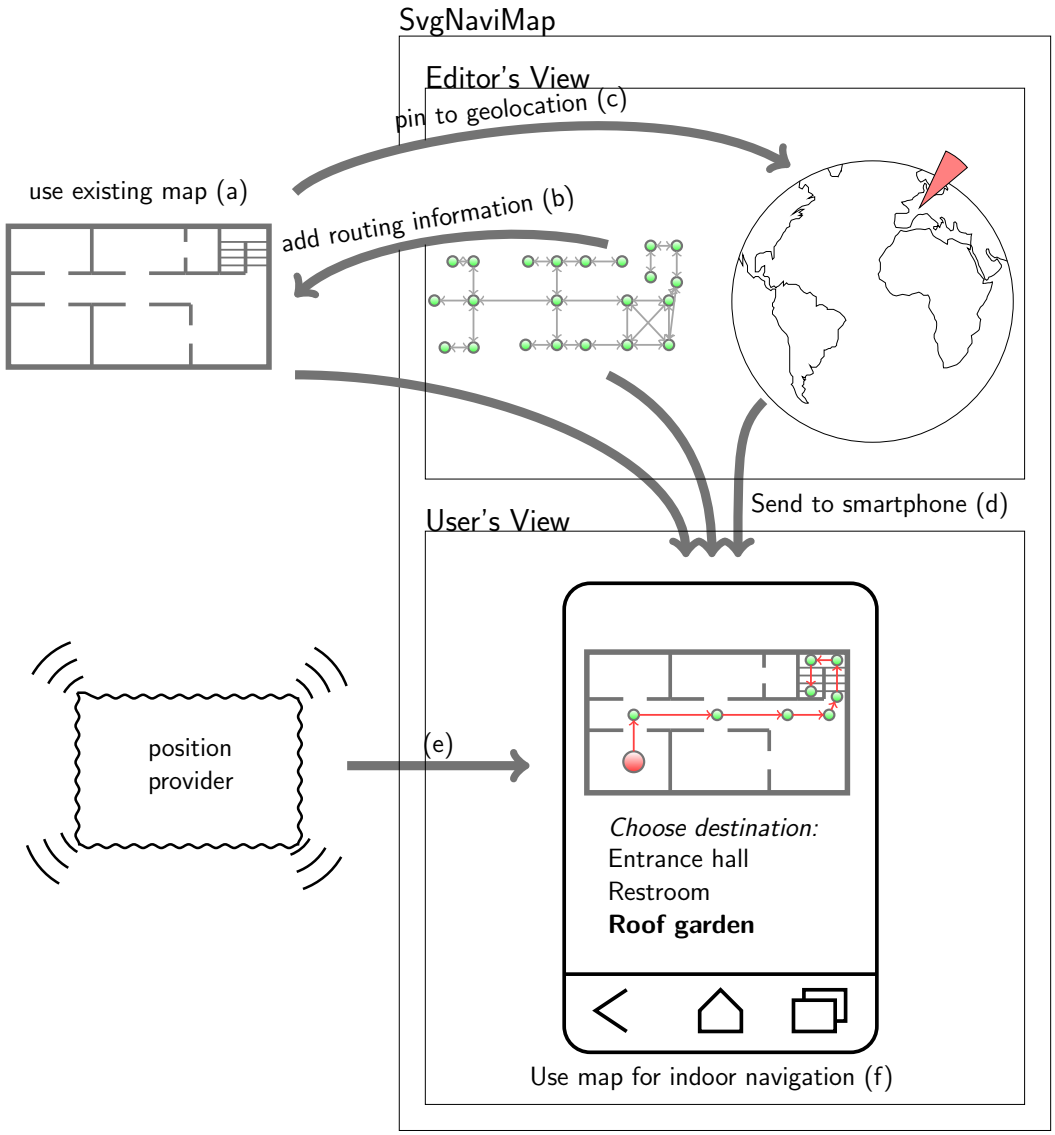


Figure 5.2: Schematic overview of how to use SvgNaviMap

In the following we present our design choices for SvgNaviMap. Afterwards we explain what kind of routing information is required.

5.8.1 Design Choices and Implications

Assuming that each large building in which a navigation system would be useful also has a server, we decided to use a distributed approach for SvgNaviMap. This way, map data is under control of the owner or IT administrator of the building in question. [...] Even though the system implements a

server-client-architecture, the web server is merely necessary for providing data. No dynamic procedures are performed on the server reducing load on the server and assuring good scalability.

As there are many mobile clients in use, and web technologies are being unified in the HTML5 standard, we chose to rely on web technologies. In specific, we designed a web application that strongly relies on SVG and JavaScript. No browser plugins are required. Route calculation is done on the client-side using JavaScript. Browsers are powerful SVG viewers which supports most features[1]. Zooming and panning is enabled by default. Further, the SVG DOM tree can be accessed and manipulated using JavaScript. For Android devices, SVG in the default browser is supported since version 3 [Chi11].

Further, it should be possible to use existing maps without modifying them. However, bitmap graphics need to be vectorized first in order to profit of the interactive features of SVG as well as lossless scaling. SvgNaviMap supports any number of levels and needs thus to support one separated SVG file per floor.

In order to be able to use any positioning technology which may become available in the future, global coordinates in the format of the World Geodetic System are being supported. Since SVG uses its own coordinate system, a mapping between the internal Cartesian SVG coordinates and global latitude and longitude is required. SvgNaviMap even supports maps which are not drawn to scale.

Navigational data is stored independently of the map data. We assume that a visitor's routing destination may be located on any floor. It is thus not practical to split routing information per level. It is not required, either, since routing information - even for a large building - is relatively small. Thus, before starting navigation, the client needs to download the complete routing information.

The actual web application is split into two independent parts. First is the Editor's View. It allows the developer to load an SVG map, then to add

[1] Overview of officially supported SVG elements in Firefox and WebKit: https://developer.mozilla.org/en/docs/SVG_in_Firefox
http://www.webkit.org/projects/svg/status.xml

and configure navigational data during setup phase. The other part is the User's View which simply displays the map and shows the current position and routing directions to the end user.

5.8.2 Routing Information

Being able to show the route, SvgNaviMap needs to have navigational information as well as the map itself. The configuration XML file combines both parts. It is thus the starting point when loading a new SvgNaviMap project. It contains links to all required SVG files. Links may either be absolute URLs or local, relative paths. Further, it contains all information required for calculating and displaying routes which is detailed in the following.

Routing is performed using a directed graph which is drawn as overlay on top the SVG map. When the user enters a destination, their location is mapped to a close vertex. From this start position routing information is displayed as arrows along the edges toward the destination vertex.

Vertices are either helper points, solely used to calculate and show routes or they are POIs. Latter are detailed by a description and can be chosen as routing destinations. They can also be grouped into self-defined categories, which allow e.g., to offer a list of all restaurants in a shopping mall. Group names are also stored within the configuration file.

The weight of each edge is by default equal to its length. Optionally, a weight factor can be manually added allowing to increase or decrease the total weight of the edge. This way e.g., stairs can be rated more expensive than an elevator; an auto walk cheaper than a normal corridor. Further, flags can be set indicating in which direction the edge is routable and whether it is wheelchair accessible. All routing information is stored in the XML format. This way more edge-specific information can easily be added.

For multistory buildings, navigation maps should provide multiple floors. SvgNaviMap is able to span the routing tree over multiple floors. Each floor is assigned a lower and upper height (floor and ceiling) allowing to map a given altitude to a floor.

Finally, for mapping between the global, geographic coordinates system and the local SVG coordinates system, two concepts are integrated in Svg-

NaviMap and its configuration file. For one, we included GPS Markers which lock an SVG position to a real-world coordinate. As floor maps are often not true to scale any number of markers are supported. For correct rotating and scaling of the coordinates systems during mapping, at least three GPS markers are required. So called affiliation areas, on the other hand, allow mapping from any SVG coordinate to one routing vertex. Thus, each affiliation area is a self-defined area around its corresponding vertex. This concept is similar to cells in a Voronoi graph, however, note that due to obstacles like walls affiliation areas cannot be computed automatically but have to be defined manually by the editor. Care must be taken in order to not overlap affiliation areas to prevent ambiguous mappings. Further, complete mapping is only possible if they cover the whole map.

5.9 Detecting and Filtering Services

According to section 5.4.2, it is assumed that MultiApp is running in background at all times. This is a major premise for autonomous service detection. In fact, MultiApp is responsible for detecting all available services (requirement {F1}), filtering for the user unappealing or unavailable services (requirement {F2}), and presenting all other services to the user (requirement {F3}).

Filtering services needs to be performed in multiple steps. The first filter should be location based: Only services which are usable at the user's current position are selectable (location filter). The next filtering step, involves checking whether the current user is legitimated to use the service (user filter). More advanced filtering steps – e.g. filtering by context – are out of scope for this dissertation.

Both, the location filter and the user filter can either be applied by MultiApp or by each service. Both options are explained in the following section and advantages and drawbacks thereof are discussed.

5.9.1 Applying Location Filter

According to subsection *Localization* in section 4.3.1, MultiApp calculates its user's position autonomously. Each service can limit its detection area according to section 5.6.3.

For MultiApp applying the location filter, it queries for *all* services in the building. Each service answers and provides its detection area. MultiApp then filters out all services which are not intended to be used at the current position.

Alternatively, MultiApp queries for all services *at the current position*. Thus, the user's position has to be transmitted to all services. Each service only responds if the user's position is within its detection area.

The first approach, has the advantage that location privacy is ensured by design. In order to avoid broadcasting interceptable user positions within the network, homomorphic encryption or a similar approach for location privacy would have to be applied [ZGH07]. A drawback of the former approach is an increased network traffic compared to the later, as *all* services respond but only a fraction of the replies is relevant. Note however, that no privacy issue is introduced, as all services which respond are public and supposed to be accessible at some location. The drawback can be amended by introducing an assumption:

Assumption 2. *All services are static. That is, services are always online and reachable at a fixed IP address and port.*

Applying assumption 2, services have to be queried only once within each building. This decreases network traffic rigorously compared to the later approach which needs to query for available services whenever the user moves. BLESS will thus follow the former approach, which requires each MultiApp client to store a list of all services per building and perform location filtering locally. Services are assumed to be online as soon as a location within the owning building was calculated.

5.9.2 Applying User Filter

It is assumed that each user owns a certificate which is known by MultiApp and by all services which the user may use as administrator or authorized

user. Filtering services by user has to be applied only for services which are available only for authorized users or administrators. Anonymously usable services are accessible by all users without providing authentication and hence are never filtered by the user filter.

Letting MultiApp apply this filter requires all services to respond to a discovery query. The response must contain a list of allowed users. MultiApp can then decide whether to display or hide services. Alternatively, MultiApp can include its user identification in service discovery requests. Each service will then only respond if it is usable by the requesting user.

The former approach introduces privacy and security issues. Malicious users can obtain the names, descriptions, and entry points of all services – even those which are not meant for them – and can thus detect available targets. The security threat should be kept low by hiding private service from not permitted users. This is better possible using the later approach.

Using public-private-key encryption, service discovery requests cannot be encrypted. This implies for the later approach that the user ID must not be sent as clear text in order to avoid privacy issues in case network traffic is sniffed. Instead using a one-way hash function the hashcode of the user ID may be transmitted. This way an attacker who is sniffing discovery request packets can match multiple requests to a single user. However, the real identity of the user stays hidden. A major drawback remains: An attackers can replay the request to discover services which are not meant to them.

Using cryptographically advanced keyed hash functions such as summarized in [BSNP+95], solves this problem: The client app generates a random key for each service request and uses this key to hash its user ID. By sending the keyed hashcode together with the key, each service can determine whether the requester is a permitted user. This however, requires each service using the given key to calculate the hash of each permitted user on each received service discovery request. This facilitates denial-of-service attacks.

As a compromise instead of a random key the current hour or day could be used. This way each service has to calculate hashcodes for all allowed user IDs only once an hour or day, respectively, and can store them. This

way replay attacks can only be performed within the chosen time slot. However, this requires that the clocks of servers and clients are synchronized. Further, replaying a service request needs to be performed only once to discover the existence and the permanent address of a service. Even the shortest time slot may thus be sufficient for an attacker.

As all proposed attempts to hide the existence of authorized services from unauthorized users have a major drawback and as user privacy is considered more important than hiding services, the very first approach will be applied: MultiApp is responsible for filtering for authorized services. All services will thus respond to all service discovery requests. Responses to discovery requests contain a list of hashed user IDs which are allowed to install the service. MultiApp filters out all services which are not meant for its current user. This way the part of user filtering of requirement {F2} is fulfilled. All services further must fulfill following assumption:

Assumption 3. *All services are secure by design. Services must not rely on hiding or obscuring their existence to provide (additional) security. They must only allow authorized users to download its CP and use its offered services.*

5.9.3 Discovery Protocol

A positive side-effect of above choices to rely on MultiApp for filtering services, is that the service discovery protocol – which is introduced in the very next section – does not need to provide support for filtering on protocol level.

Existing Discovery Protocols

Current discovery protocols for IP networks are: SSDP, DNS-SD/mDNS, and SLP as described in section 2.4.2. Ignoring the fact that SLP requires root permission, all of them could be used use for BLESS. Unfortunately, none of them is considered ideal.

The common disadvantage of SSDP and the combination of DNS-SD and mDNS (m/DNS) is their complexity. While SSDP requires elaborate XML

handling and needs to be extracted from the UPnP protocol stack, m/DNS uses a complex workaround to allow service discovery via standard DNS queries. SLP on the other hand is much simpler. However, it cannot be applied on non-rooted smartphones.

Considering mDNS opposed to SSDP and SLP in multicast mode, it uses a link-local multicast address. It can thus not be used in a network consisting of multiple subnets. In this case DNS-SD would be required. However, like SLP with service directory, it introduces a single-point of failure which contradicts the vision of an ideal services system from section 3.3. Additionally, there is only a single implementation of a fully compatible DNS-SD server available which is most likely not present in most networks and needs thus to be set up, complicating the adoption of BLESS.

A drawback of all three of them is, that responses to search queries do not contain all relevant information about the service. For example for SSDP a dedicated XML file has to be requested while SLP needs an explicit request for obtaining the service's attributes. Another common inadequacy for location-linked services is the timeouts for service announcements. Libraries of discovery protocol are usually implemented such that they delete services from the internal discovery list if the corresponding announcement is not renewed. As this is consistent with the protocols, libraries do not offer a method to circumvent deletion. Services which are not listed cannot be accessed. Consequently, if any of the considered discovery protocols was used, service announcement would have to be repeated. However, according to section 5.9.1 services are static, i.e. they do not appear and disappear frequently. Repeated announcements and service discovery queries are thus not required and would unnecessarily increase network traffic.

For BLESS a simpler discovery protocol is preferable. It is specified in the following section.

Discovery Protocol Specification for BLESS

As described above none of the existing discovery protocols are suited for BLESS. Since BLESS does not require communication with any existing, external entities, a custom protocol may be incorporated without introducing incompatibilities. The protocol presented by this dissertation is kept as sim-

ple as possible and avoids generalization, to allow easy adaption by future BLESS services. Learning from the analyzed protocols, following characteristics are required:

- Queries: Find all BLESS services, find all BLESS buildings with sub-buildings, find BLESS pusher (cf. section 5.10), find all three types at once
- Query responses include all details of service or building
- No timeout for discovered services
- Multicast address from *Administratively Scoped Block*
- Listening port above 1024

For service discovery of discoverable BLESS entities, following simple protocol specification will be observed. Searching entities will be called *clients* while *servers* answer to query requests.

Clients send multicast search requests as UDP datagram to multicast address `239.255.81.35` and port `8135`. Content of the datagram is UTF-8 encoded. The first character specifies and query type: `S` for BLESS services, `B` for BLESS buildings and sub-buildings, `P` for the BLESS pusher, `A` for all types. Separated by a colon, follows the port to which the response is to be sent to as visualized in figure 5.3.

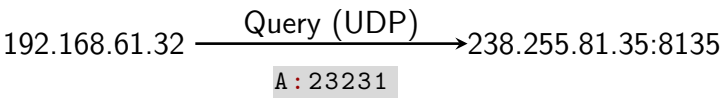


Figure 5.3: Example of query for all BLESS entities

Responses are considered important and must not be lost. Further, they are usually too big to fit into a UDP datagram. Thus, all servers receiving a query request for its type, respond via unicast to the sender by sending a TCP packet to the provided port. The content is a UTF-8 encoded JavaScript Object Notation (JSON) object. It must contain following fields: `type`, `id` (identification), and `entrypoint`. Content of `type` is: `service` for BLESS service, `building` for BLESS building, `sub-building` for BLESS sub-building, `pusher` for BLESS pusher. Field `id` must contain a unique identifier for the BLESS entity. Field

`entrypoint` indicates the destination address at which the server provides its functionalities. For services it is the path to the remote interface, for buildings the path to obtain building data, and for the pusher the path where it is listening for incoming connections. For all three cases the path may be preceded by a colon followed by the TCP port.

Both building types must provide additionally fields `name` and `area`. Former is a friendly name which is shown to the user. Latter is a sorted array containing at least 3 geographic coordinates in decimal degrees format defining the area of the building. Latitude and longitude are separated by a comma and use a dot as decimal separator. Northern latitude and eastern longitude are stated as positive numbers. The coordinates indicate the vertices of the building. Further, sub-buildings must specify the ID of its parent building via field `parent`. An example of a response of a building and a sub-building is shown in figure 5.4.

192.168.61.32:23231 ← Building (TCP) — 192.168.61.35

```
{
  "type": "building",
  "id": "bliss.building.tuhh",
  "name": "TUHH Campus",
  "area": [
    "53.463897841281856,9.971320925598093",
    "53.45842992675125,9.971320925598093",
    "53.45842992675125,9.967469300552466",
    "53.463897841281856,9.967469300552466"
  ],
  "entrypoint": ":8080/tuhh.zip"
}
```

192.168.61.32:23231 ← Subbuilding (TCP) — 192.168.61.35

```
{
  "type": "subbuilding",
  "id": "bliss.building.tuhh.sbs95e",
  "parent": "bliss.building.tuhh",
  "name": "TUHH SBS95E",
  "entrypoint": ":8080/buildings/e.zip",
  "area": ...
}
```

Figure 5.4: Example response of a building and a sub-building

Also, BLESS services must provide field `name`. Additionally, they must define a `description` for the entity which is shown to the user. With these two fields the requirement of providing service details {F4} is fulfilled. Restricted services for authorized users only must define field `users`. It is an array containing hashes of authorized user identifications used by MultiApp for the user filter. As one-way hash function SHA3-256 is proposed.

Optionally, if a service belongs to a sub-building rather than a building, field `subbuilding` is required. It must either contain the identifier of or a geo coordinate inside a sub-building which belongs to the current building. If the area of usage of a service is to be limited also within the sub-building, field `rooms` needs to be defined. It is an array containing names of rooms in which the service should be detected. Note that the names of the rooms must be those also used by the localization data provided by the building service.

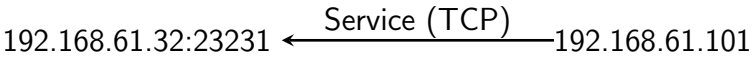
An example of a response of two services is provided in figure 5.5. Note that the first service is linked to a building because no `subbuilding` is defined, in this case building `TUHH Campus` (from Figure 5.4). The latter service belongs to sub-building `TUHH SBS95E` and is only intended to be used in the defined rooms.

Clients should listen for several seconds (e.g. 10) for server responses before closing the listening port. Clients should issue query requests when entering buildings. Periodic queries must be avoided. Manual queries can be supported. All variable names and strings are case-sensitive. Servers may ignore requests from clients which query too frequently (e.g. more often than once in two minutes).

For completeness sake, figure 5.6 shows an example of a response sent by a BLESS pusher entity.

5.9.4 Enabling Service Detection

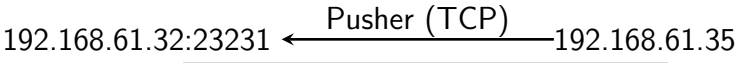
Discovery of services is closely linked to localization, because BLESS services are only available within buildings, that is within WiFi networks. Whenever localization determines that the user is outside any building, service




```
{
  "type": "service",
  "id": "bliss.service.tuhh.navigator",
  "name": "Navigator",
  "description": "Find your way at TUHH",
  "entrypoint": "/svcs/navigator.zip"
}
```


```
{
  "type": "service",
  "id": "bliss.service.tuhh.sbs95e.telematik.lights",
  "name": "Light Control Telematics",
  "description": "Switch lights in Telematics offices",
  "entrypoint": ":81/lcTelematics.zip",
  "subbuilding": "bliss.building.tuhh.sbs95e",
  "rooms": ["4.075", "4.076", "4.077", ..., "4.091"],
  "users": ["dbd487b601282fc86308b4f5f4b283c7af0fa97db
     64b3f6f0c0c47f472c61d26"]
}
```

Figure 5.5: Example response of a BLESS service




```
{
  "type": "pusher",
  "id": "bliss.pusher.tuhh",
  "entrypoint": "3265"
}
```

Figure 5.6: Example response of BLESS pusher

discovery cannot yield any results and should thus be disabled. On the other hand, when service discovery detected a BLESS building, the localization algorithm is expected to be able to determine a position. It should be a position within or close to the building.

This interconnection is most important for detecting when the user enters a BLESS-enabled building. In this case the user needs to be informed about available BLESS services as explained in section 3.3. However, neither periodically searching for building services nor continuous location scans are appropriate, as both procedures require processing power and use the WiFi adapter. They will thus quickly drain the battery. Decreasing the scan inter-

val results in worse performance in terms of speed of detecting buildings.

Fortunately, modern smartphone operating systems provide means of detecting changes in the WiFi connection. Apps can thus be informed when the WiFi adapter connects to or detects a new network. The actual detection is thus delegated to the operating system. As detecting WiFi networks is a fundamental part of mobile operating systems, it is highly optimized.

MultiApp will thus listen to WiFi network changes and start a discovery request when connected to a new network. The results need to be stored to avoid unnecessary, repeated scanning of networks. If the WiFi connection is lost, it is likely that the user left the building. This should be confirmed by using the localization algorithm to avoid false location events due to poor WiFi coverage inside the building.

5.10 Inter-Service Communication

For BLESS two communication patterns between MultiApp and services are required as specified in section 4.3.1. First, the client must be able to invoke actions which includes retrieving data from the server. Second, via the pusher the server needs to send notifications which in turn may invoke actions on the client. Further, it has to be observed that the CP is written in JavaScript while the language for the SP is not specified.

5.10.1 Method of Communication

As specified by the requirements, a two-way communication between SP and CP is mandatory. According to section 2.6.1, there exist currently four options for pushing messages to smartphone clients. This dissertation will apply the approach of keeping a single open connection from smartphone to a third party relay service. It promises best scalability and is less resource consuming for smartphones while not depending on a special network configurations. However, for privacy, security, and connectivity reasons BLESS must not rely on the push message mechanism offered by the mobile operating system. Instead BLESS introduces a custom push messaging mechanism.

To avoid requiring an Internet connection, each building must provide such a relay service. It will be called *pusher* in the following to circumvent overloading the term *service*. The pusher must implement the same discovery protocol as BLESS services. This allows smartphones as well as services to detect the relay service in order to use it without configuration. The smartphone identification is to be sent by MultiApp to the SP of a service when necessary.

This approach introduces a single point of failure for communication from server to client. This is according to the design requirement {N1} to be avoided. However, the alternative would be either resource consuming on the smartphone part or would require a special network configuration as described in section 2.6.1. Further, without a pusher acting as relay service, the server part of each service would have to implement message caching and re-sending mechanisms in case clients are temporarily not available. Thus, this single point of failure is deemed acceptable considering the drawbacks of alternatives. Further, services can implemented a poll mechanism as fallback option.

For the opposite direction, i.e. for messages sent by clients to servers, any state-of-the-art communication protocol over TCP/IP may be used.

5.10.2 Server Methods Invocation Protocol

As creating services shall be kept easy for developers, pure interprocess communication as described in section 2.6.3 is due to its development efforts not an option. Also indirect communication is not an option as it would introduce unnecessary complexity into the BLESS framework. The remote invocation approach, however, fulfills the requirements well.

For BLESS there two options available to implement remote invocations on the client-side, i.e. within MultiApp. Either the CP itself can perform the remote method invocations or an indirection layer could be introduced. Latter approach would mean that MultiApp determines which remote interfaces are offered by the server part and makes according JavaScript interfaces available to the client side. Marshaling and communication would then be the responsibility of MultiApp. This approach, however, comes with a num-

ber of disadvantages: The JavaScript interface for remote calls would have to be created dynamically during runtime. This would increase the complexity of MultiApp and further, the remote interface would not be available during design time, hampering development. The indirection layer might also negatively impact performance. Further, it is not required because also JavaScript code can directly contact the server part either using AJAX calls or by using the WebSocket technology. Thus, the first approach is to be considered more appropriate and will be followed by this dissertation.

Today there are many protocols and implementations of remote invocations available. In general two kinds of RPC projects can be distinguished. The first type aims primarily to define complete protocol specification, fixing the byte sequences which are to be transfered between client and server. This is often done in form of an IDL (Interface Description Language). The specification then may be implemented using any programming language on any operating system. Prominent examples include JSON-RPC, XML-RPC, Common Object Request Broker Architecture (CORBA), Simple Object Access Protocol (SOAP), and UPnP. For these protocols many implementations and helper libraries in different programming languages are available.

The second type of RPC projects primarily aims to provide an RPC framework to be used by programmers. Usually these projects also provide a detailed protocol specification, however, for a programmer it is intended to be of minor interest. Projects of this type include Apache Thrift[1], Java RMI[2], and ZeroC ICE[3].

To reduce development efforts for BLESS, in a first step projects of the second type were evaluated. Unfortunately, none of them can be applied directly. The major issues are that JavaScript is not supported, the documentation is incomplete or incomprehensible, or both. ZeroC ICE was discarded due to its complexity to set up. A primary goal is to keep development of services as easy as possible. An integration into the BLESS framework would - besides making BLESS itself unreasonable complex - also presumably hamper the work of service developers.

[1] https://thrift.apache.org/

[2] https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/

[3] https://zeroc.com/products/ice

Of the first set of projects CORBA was discarded because it is too generic and as a result very complex. UPnP could not be used due to several architectural shortcomings, including: While discovering services it is very verbose (cf. section 2.4.2) and notifications are always broadcasted impeding private messages to single users (cf. section 2.6.1).

XML-RPC and SOAP which both rely on the Extensible Markup Language (XML) for formatting messages are not used due to the verbose nature of XML. Instead in this dissertation remote invocations will be marshaled applying the JSON-RPC format. It was chosen as protocol for exchanging remote invocations message mainly due to its simplicity. It further was considered as a good choice because it uses JSON encoding format which is natively supported by JavaScript engines and is less verbose than the XML format. Better performance on client-side is thus to be expected due to favoring JSON over XML. Consequently also for serializing method arguments and return values JSON will be used.

Using JavaScript the communication module can be either based on asynchronous HTTP calls or the WebSocket technology. Considering the current HTTP 1.1 standard enables the keep-alive option by default, both approaches reuse open TCP connections. However, using WebSockets it is possible for the server to push messages to the client. For this reason in this dissertation the recent WebSockets technology will be applied but HTTP is used as fallback.

5.10.3 Push Messaging Protocol

As motivated in section 5.10.1 for push messages from server to client a custom messaging protocol is introduced. It involve three entities. Their interaction is shown in figure 5.7. Table 5.1 provides an overview of the packet types of the protocol.

A running CP can inform MultiApp at any time, that it is expecting push messages. MultiApp will then register itself with the SP as well as with the pusher which belongs to the same building as the requesting service. Registering involves sending a unique device identifier to the pusher and the SP. Former is part of this protocol. Later, i.e. passing the device identi-

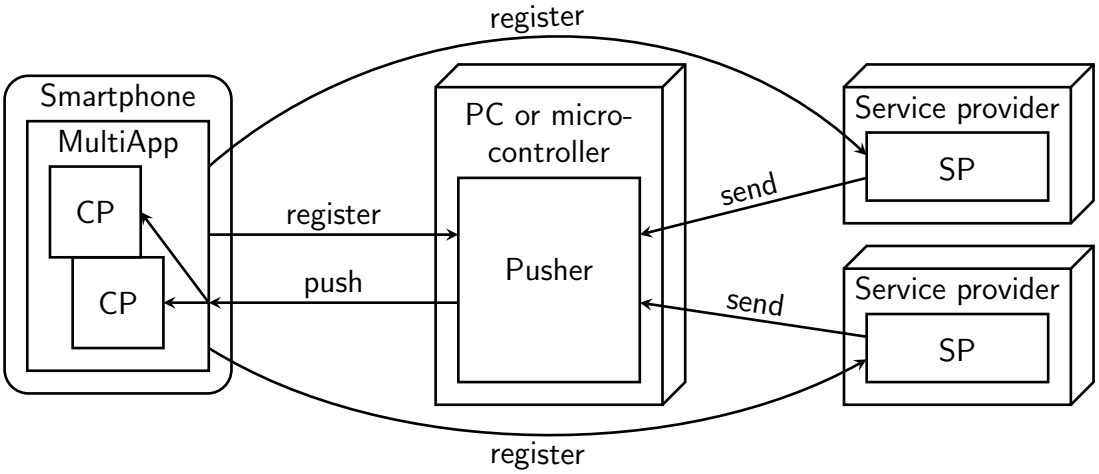


Figure 5.7: Overview of BLESS push messaging architecture

fier to the SP, is accomplished via a service method invocation. This way, the SP does not need to open an additional server socket, helping to keep complexity of the SP low.

Further, MultiApp establishes a permanent TCP connection to the pusher. If the connection breaks it is re-established as long as at least one CP belonging to the current building is registered for push messages while being connected to the according WiFi network. For detecting broken connections which are not reported by the TCP layer a heart beat named *awake beacon* is proposed. It is sent regularly by the pusher to each client with the first being the registration acknowledgment. Clients which do not receive the beacon in time, must assume their connection to the pusher is broken and must thus try to reconnect.

At this point a SP can send a forward request to the pusher. It contains as payload the message which is to be pushed to the client as well as the device identifier. If the message is signed with the service's certificate the origin of the sender can be reliably identified by the pusher. Only if the sender is trusted, the message is forwarded to the client via the TCP connection corresponding to the identifier. Additionally, a service identifier is included. If a signature is used, it is redundant and should be empty. If the connection is down, the message is stored later delivery. The device identifier generated by MultiApp is stored persistently and reused for all

future registrations. This ensures that messages sent from a SP to the pusher can be delivered when the connection between MultiApp and pusher is re-established.

A message received by MultiApp may be checked again using the signature of the message. Only messages from the SP belonging to the same service are delivered to the CP.

Packet name	Contained data	Sender → receiver
RegAtPusher	clientId	MultiApp → Pusher
SP2Pusher	signature, clientId, serviceId, payload	SP → Pusher
Pusher2CP	serviceId, payload	Pusher → MultiApp
AwakeBeacon		Pusher → MultiApp

Table 5.1: Overview of packet types of custom push messaging protocol

5.11 Assumed Preconditions

This section summarizes all preconditions which need to be fulfilled in order to apply this design.

In order to be able to use the introduced service provisioning systems, MultiApp must be installed by all users. Further, it is assumed that it is running at all times so that it can detect available BLESS entities. Buildings offering BLESS services must be equipped with APs to provide complete WiFi coverage. It is assumed that users are connected with the local network.

For user authentication each user requires a personal certificate. The public key must be known to servers which need to authenticate the user. For authenticating services the users must trust the certificate of the author of the services. For public buildings trusting the building's root certificate is assumed (cf. section 5.5.1).

Chapter 6

Prototype of BLESS: MultiApp and Exemplary Services

This chapter describes the prototypical implementation of service provisioning system BLESS, fulfilling the requirements from chapter 4 by implementing the design from chapter 5. The goal is not a complete implementation of the design but rather a system which can be used to evaluate the design. This chapter provides details about implementations of all major software components.

6.1 Overview of BLESS Implementation

The BLESS architecture comprises five different software component types: MultiApp, services, pusher, buildings, and sub-buildings, whereas the last two are very similar. There are further four different protocols used for communication between component types. Their interaction is graphically presented in figure 6.1. Details about each software component and the protocols are presented in the following sections.

The functioning prototype of BLESS created in this dissertation comprises – next to MultiApp and pusher – five services and three buildings including sub-buildings. For all components the design choices from chapter 5 were regarded best possibly. Simplifications in order to reduce implementation efforts were only made if they do not affect functionality. Omissions of design details were only allowed for parts that were considered

pure routine pieces of work.

The result is a functional prototype of the BLESS system. Its Android client application is mainly missing secure authentication and user permissions. Still the prototype demonstrates the general soundness of the design.

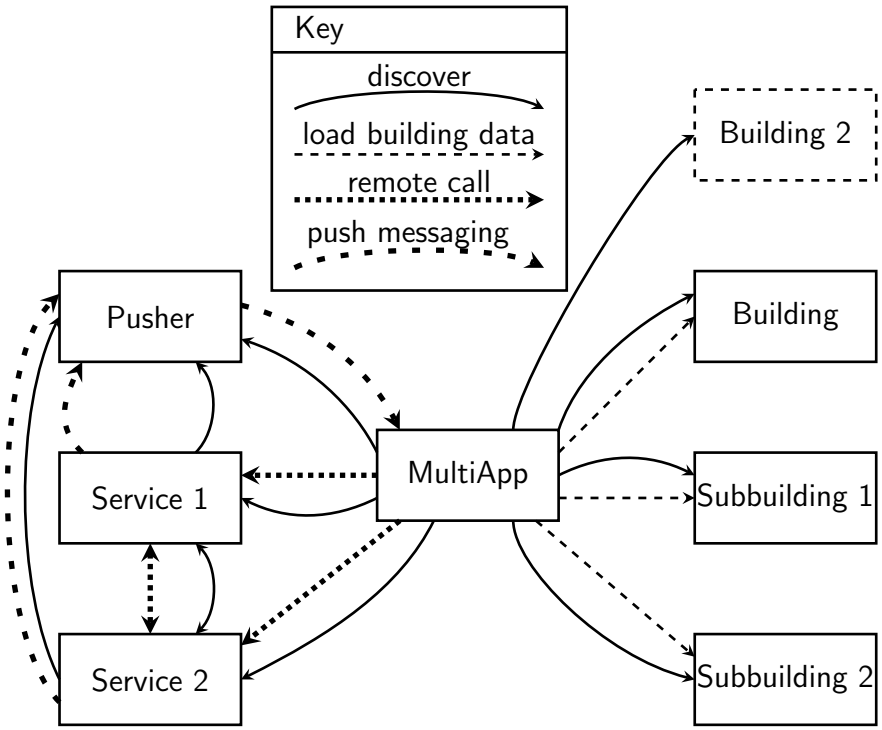


Figure 6.1: Architectural overview of BLESS

6.2 Implementation of MultiApp

MultiApp is the most complex component of BLESS. It was implemented as Android app only. However, there is no technical reason which would hamper implementation for other mobile platforms. Its main internal components are presented in the following.

6.2.1 Discovering BLESS Entities

MultiApp employs two discovery protocols for detecting BLESS entities in local networks in order to fulfill requirement {F1}. In a first approach UPnP

discovery was used by deploying open source library Cling[1]. Afterwards the m/DNS approach presented in section 5.9.3 was implemented relying on mDNS library JmDNS[2]. The according unicast protocol DNS-SD was tested as proof-of-concept by setting up a BIND DNS server[3] which supports dynamic DNS updates. The client-part of DNS-SD for creating and deleting DNS entries was implemented using library dnsjava[4]. However, DNS-SD was not integrated in MultiApp. Neither the custom discovery protocol proposed in section 5.9.3 was implemented, as it is only designed to reduce network traffic but does not provide additional functionalities. All protocols offer a field for a friendly name which is used as name for the BLESS services to partly fulfill requirement {F4}. However, only the custom protocol provides means to offer a textual description for services.

As required by requirement {F19}, CPs of services which registered for online events are notified whenever their corresponding SP is discovered or becomes unavailable.

6.2.2 Signature Handling

The specification of BLESS proposes that certificates are used for authenticating entities as well as users (cf. section 5.5.1). However, as authentication via certificates is already state-of-the-art, no proof-of-concept is required for the prototypical implementation of BLESS. Thus, signature handling was implemented in a strongly simplified manner.

Instead of signing their communication packets, entities and MultiApp append the author's and user's e-mail address, respectively, as placeholder for the certificate. MultiApp and each service thus manage a list of e-mail addresses instead of public certificates. Upon receiving a message the included e-mail address is used to uniquely identify the communication partner. Complex cryptography can thus be avoided and facilitates implementation of MultiApp and services. However, no reliable authentication can be provided by this approach and must be substituted after prototyping in

[1] http://4thline.org/projects/cling/
[2] http://jmdns.sourceforge.net/
[3] Berkeley Internet Name Domain: https://www.isc.org/downloads/bind/
[4] http://www.dnsjava.org/

order to fulfill security-related requirements {F5}, {F6}, {F7}, and {N5}.

6.2.3 Listing and Filtering of Available Entities

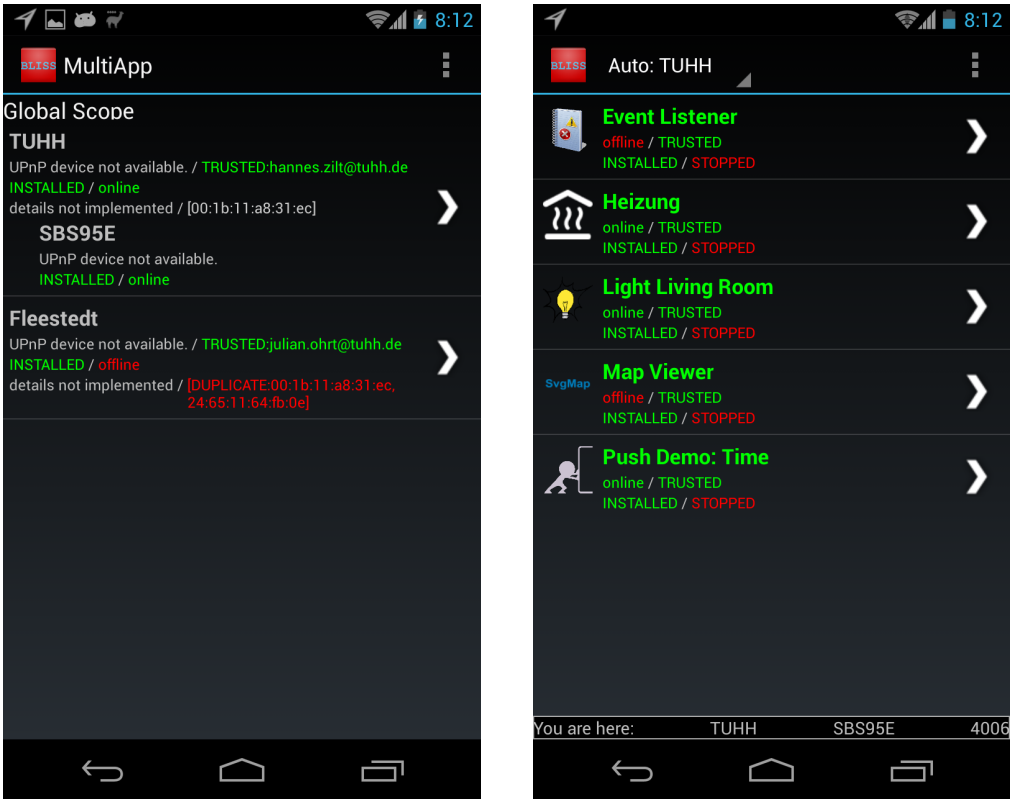
MultiApp manages separate lists for BLESS services and buildings together with sub-buildings. Both lists can be displayed in the GUI. They show which entities are online, trusted, and installed. For services also the running state is displayed. On clicking an entry further details are displayed. Room-wise filtering as envisioned by requirement {F2} and section 5.9.1 was not implemented due to the fact that the used localization technology was not accurate enough. Erratic list changes would have been the result. Instead all services are displayed which belong to the current building. The current building can either be detected automatically using the integrated localization algorithms and by manually choosing a building. Screenshots are provided in figure 6.2.

Since signature handling was implemented in a strongly simplified manner, filtering services according to the current user as described in section 5.9.1 could not be realized in a useful way. It was thus not implemented.

6.2.4 Installation of Buildings and Services

Discovered buildings and services which are online may be installed by the user according to section 5.4.1. Untrusted entities can only be installed after accepting an according warning dialog. During installation of a building the offline localization database and the maps are loading and stored persistently in internal memory.

Installing a service involves loading and storing the CP. It is thus possible to launch all services while their SP is not available, fulfilling requirement {F11}. Further, during installation a mapping between service and current building is created. It allows to filter services which do not belong to the building being currently inside.



(a) Buildings list (b) Services list

Figure 6.2: Screenshots of MultiApp showing lists of available buildings and services

6.2.5 Executing CP of Services

After installation, services can be launched. This involves opening a full screen WebView element in which the web application contained within the CP is displayed. This allows creating very flexible GUI fulfilling requirement {F21}. For the WebView implementation of MultiApp relies on Apache Cordova[1]. Cordova provides a number of plugins which allow a web app – in this case the CP – to interact with the hosting operating system by providing appropriate programming interfaces. Following requirements are fulfilled this way: Playing audio files ({F22}), accessing the accelerometer sensor (requirement {F24}) and getting the orientation from compass (requirement {F25}).

Further, MultiApp offers an additional plugin for the Cordova framework. It implements interfaces accessible from the CPs' JavaScript code for following requirements: Obtaining details about the current building allowing to access map and navigation data (requirement {F26}), accessing the current user position (requirement {F27}), issuing notifications using the system's native notification bar (requirement {F23}), registering for server push messages (requirement {F29}), and allowing services to invoke other services and to share data with them (requirement {F30}).

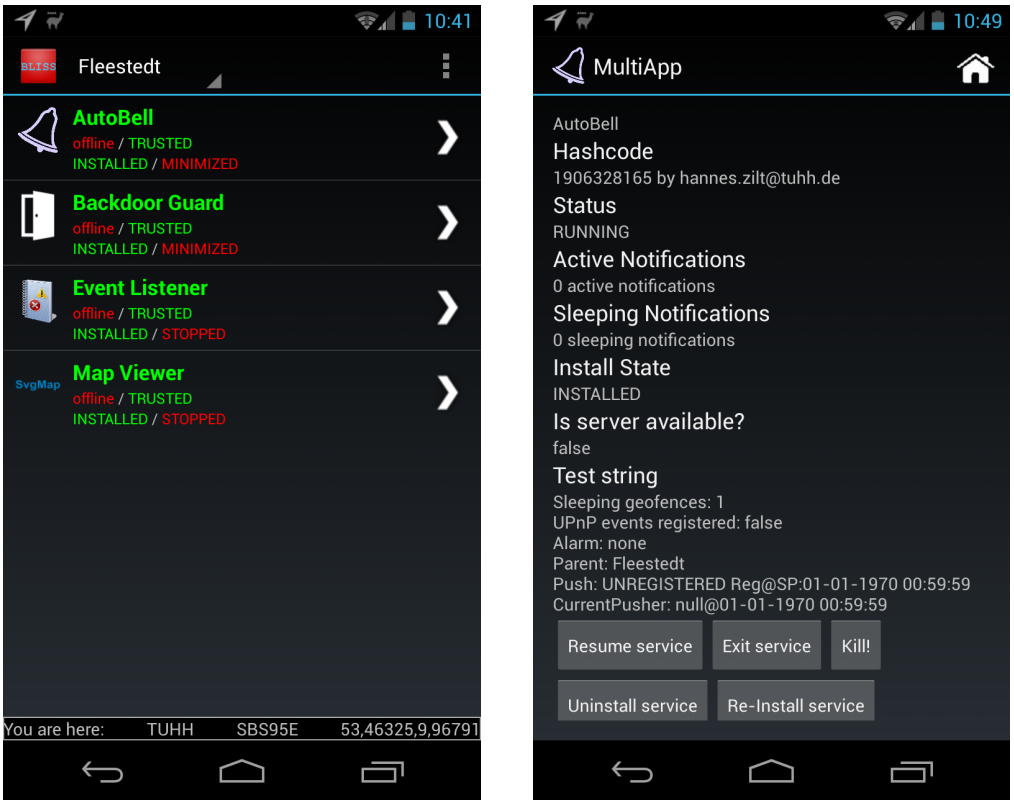
Storing private data can be achieved directly using the LocalStorage which is part of HTML5 (requirement {F31}). Also the client to server communication (requirement {F28}) relies on built-in JavaScript objects, which are `WebSocket` and as fallback `XMLHttpRequest`. The entry point for the remote connections, i.e. the address of the SP of the service, is provided by MultiApp.

6.2.6 Sending CP into Background and Wake Events

The Android implementation of MultiApp realizes the three different running states of services as described in section 5.4.2. This way requirement of running services in background {F20} is fulfilled. For executing services a single WebView element is used. This implies that only one service can be in *running* state at a time.

[1] https://cordova.apache.org/

All wake events from section 4.3.2 (requirements {F14} throughout {F19}) were implemented. While a user and another service can launch a service at all times (requirements {F14} and {F15}), the other wake events need to be enabled by the service itself. For each a corresponding JavaScript method is offered by MultiApp.



(a) Services list with minimized services (b) Details view of service

Figure 6.3: Screenshots MultiApp

In the services overview list the user can see which services are currently minimized as shown in figure 6.3a. Via the service's details view – which contains debug data in the prototype as shown in Figure 6.3b – the user can kill the service which unloads the service from the WebView if it is currently running and additionally removes all wake event registrations. Thus after killing a service it always is in *stopped* mode.

6.2.7 Invoking Remote Calls

For invoking server methods (requirement {F28}), the JSON-RPC protocol as described in section 5.10.2 was implemented. As the server side is implemented in Java, a compiler was created which creates from a Java remote interface a JavaScript stub which handles marshaling of remote calls and corresponding responses. The stub can now be accessed by the JavaScript code of the CP and directly create a connection to the SP. The stub can also be used during design phase to enable code completion within integrated development environments. All these functions are provided by a service skeleton library which is presented in section 6.6.1.

Before any remote calls can be issued, the remote address of the server needs to be set. It is published by the service during discovery process (cf. section 5.9.3) and is thus known by MultiApp. A CP which needs to communicate with its SP needs to register for remote calls using the appropriate interface offered by MultiApp. MultiApp in turn forwards the remote address to the JavaScript stub which then becomes able to invoke remote calls.

6.2.8 Push Notifications

CPs of services that need to receive push messages from its SP according to requirement {F29}, need to register using the appropriate interface offered by MultiApp. MultiApp then ensures that push messages can be sent and received as described in section 5.10.3. For creating the permanent TCP connection the free library PyroNet[1] is used. The device identifier is transmitted using the appropriate remote interface offered by the SP.

6.2.9 Localization

Due to the lack of an available, directly applicable indoor positioning technology, four custom localization algorithms were implemented within the localization framework of MultiApp to fulfill requirement {F27}.

[1] https://code.google.com/p/pyronet/

The first algorithm relies on BLESS buildings and is self-learning. While a building is detected to be online, the BSSIDs of each connected WiFi access point is stored. When the mobile device connects to a known access point again, a simple look-up function can determine which building was just entered even before the building service was detected. Building-wise granularity can thus be achieved almost instantaneous. This localization technique is called BSSID-based localization in the following

The second approach works similarly, however, an existing database is required and is called WiFi DB localization. The database contains BSSIDs and their geo coordinates. By looking up currently available BSSIDs in the database the current position can be approximated. An appropriate database must be made available by BLESS buildings. While there exist many such privately owned databases[1] there are also some well documented and query-able datebase available on the Internet[2]. As part of this dissertation several tools were developed for extracting data from these source, for collecting own data, and for converting data into appropriate formats. By importing data into a SQLite database positions can be looked up within a few milliseconds with a granularity of 10 m to 50 m.

The last two algorithms apply machine learning techniques Random Forest as introduced by [Bre01] and Hyberbolic Location Fingerprinting presented by [KM08]. For both algorithms existing and freely available libraries were used[3]. Appropriate fingerprint databases were created using a custom application based on the SvgNaviMap project. Both algorithms achieve room-based granularity which provides correct results in about 50% to 90% of all test cases. While the calculation time for both algorithms heavily depends on the size of the database, the former algorithms tends to be much faster and delivers the first result in less than 10 s and subse-

[1] e.g. Apple: https://support.apple.com/en-us/HT201357,
Google: https://developers.google.com/maps/documentation/geolocation,
Mozilla: https://location.services.mozilla.com/

[2] Examples include https://wigle.net/, http://openbmap.org/, and http://www.openwlanmap.org/.

[3] Fast Random Forest is available here: https://code.google.com/p/fast-random-forest/, Hyberbolic Location Fingerprinting implemented as part of SmartCampusAAU: https://github.com/BentThomsen/SmartCampusAAU

quent results in less than half a second.

If the user position has to be obtained, MultiApp always determines the current building using the BSSID-based algorithm first. Next, the second approach is pursued if the current building provides an according WiFi database. Of the last two algorithms only the Random Forest was used due to its advantage in terms of speed while providing similar accuracy. If according fingerprint databases are available, this last algorithm is either executed for the current building or, if sub-buildings are available, iteratively for all sub-buildings until a position was determined. However, it is only executed if the second approach determined a position inside or close to the according building or sub-building. It was empirically determined that an appropriate value for determining close buildings is a radius of 50 m.

6.2.10 Geofencing

MultiApp offers a geofence interface. Similar to Android's Geofence API (cf. section 2.2.2), it allows the CP of services to register for events which indicate when the user enters or leaves a given area as envisioned by requirement {F16}. It further enables events when the user dwells inside or outside an area for a given duration of time.

Three types of geofence areas are supported: Collection of rooms inside a given building, a geographical area defined by geo coordinates, and whole buildings. For the first type a room-wise location is required. Thus the machine learning localization algorithm has to be applied. The second area type can be determined using WiFi DB localization. Detecting whether the user is inside or outside a building can be achieved using BSSID-based localization. Depending on the installed geofences only the corresponding localization algorithms are executed.

For performance and energy consumption reasons, machine learning localization is only applied if a close position was determined using WiFi DB localization. It was empirically determined that an appropriate value for determining closeness is a radius of 500 m. The scan interval is configured dynamically according the current distance to the closest registered geofence. Following, empirically determined intervals are used: If the distance

is larger than 5 km no scans are initiated by MultiApp, however, all WiFi scan results are evaluated to calculate a new distance and thus update the scan interval. For distances smaller or equal to 5 km but larger than 1 km the scan interval is set to 7 min. With active geofences with a distances of equal or less than 1 km every 3 min a WiFi scan is initiated.

6.2.11 Permissions of CPs

Asking for the user's permission before a service performs security relevant actions as envisioned by requirement {N8} in section 5.5.3 was not implemented.

However, all listed actions except communication with the SP are performed via interfaces offered either by MultiApp or Cordova. MultiApp in turn also relies on Cordova to offer its interfaces. Cordova uses internally a single entry point for all JavaScript calls in Cordova's class `ExposedJsApi`. Each request from JavaScript code querying to access Cordova's functionalities is relayed through this function. It is a simple matters of extending named function to filter security relevant actions called by services not owning the corresponding permissions. For each filter hit a security alert dialog has to be displayed as described in section 5.5.3.

In order prevent communication with the SP, MultiApp needs to set all JavaScript objects which allow to establish external communication to null. These are currently the `WebSocket` and the `XMLHttpRequest` objects. Without being able to access these objects, no network communication can be performed by JavaScript code. Once a service requested and the user granted the according permission MultiApp needs to recover these objects.

6.3 Protocols

Four distinct communication protocols between BLESS entities can be identified. While their design details are summarized in chapter 5, this section provides an overview of their implementations including how they practically function.

6.3.1 Discovery Protocol

Both discovery protocols applied by BLESS – UPnP's SSDP as well as mDNS – are used in a very similar way: MultiApp sends discovery packets via TCP multicast to all discoverable BLESS entities. Querying only for a specific BLESS entity type is not possible. The receivers respond with a summarized answer via unicast. MultiApp in turn has to request further details in a separate query.

6.3.2 Installation of Buildings and Services

The installation of buildings and services is a simple download of a ZIP compressed archive. As protocol HTTP is used. The URL is provided as detail of the previously discovered building or service entity.

6.3.3 Server Method Invocation

Using the UPnP protocol server methods are invoked via UPnP actions exported by the SP of a service. The corresponding CP can invoke actions by specifying the action names and passing it together with the required call parameters to MultiApp. Since MultiApp is aware of the UPnP device it can issue the UPnP remote call.

The custom remote invocation protocol specified in section 5.10.2 requires JSON-RPC-formatted action requests. According marshaling is accomplished by the JavaScript client stubs which are created by the service skeleton at development time. The remote entry point for remote invocations is determined byMultiApp during service discovery.

6.3.4 Push Messaging

The custom push messaging protocol introduced in section 5.10.3 defines four message types:

- For sending message from service part of service to Bliss Pusher.
- For sending message from Bliss Pusher to CP.

- Register client at Bliss Pusher.
- Pusher periodically sends awake beacon to clients. First beacon is sent as registration confirmation.

6.4 BLESS Pusher

The BLESS pusher was implemented as Java application. For realizing the push messaging protocol as described in section 5.10.3, again library PyroNet was used. Service discovery was enabled both using UPnP library Cling and also mDNS library JmDNS.

At this point no authentication is implemented. However, the service's signature is sent to the pusher which is displayed on the console for debugging and testing reasons. Implementing filtering based on the sent signature for this prototype would not bring additional benefits.

Due to the generic nature of the push messaging protocol, the pusher implementation can be used as is in any number of buildings.

6.5 BLESS Buildings and Sub-Buildings

For this dissertation building service for the university campus of the TUHH and a sub-building service for physical office building E of the TUHH was implemented. Additionally, a building for the author's home was created. These three entities are called TUHH, Office, and Home, respectively, in the following.

For TUHH and Home databases for WiFi DB localization were created covering the buildings and also the surrounding neighborhood. Data was taken from the Wigle project and complemented with self-collected WiFi data. The two databases include 25211 and 10569 positions of WiFi access points, respectively. Fingerprint databases were created for Home and Office, covering the whole Home and all corridors of Office, respectively. Further, for all three structures, building maps including navigational data were created using SvgNaviMap. For Home these maps include three floors with 26 rooms and relevant positions outside. The Office map covers five

floors with a total of 480 rooms. The fingerprint database for Home includes all 26 positions. Due to the large building size of Office and due to access restrictions, only 102 position were mapped via WiFi fingerprinting. The resulting file and database sizes are presented in section 6.7.3.

The implementations themselves again rely on earlier mentioned libraries for providing UPnP and mDNS discovery. Creating new buildings is a simple matter of configuration. First the map, navigation, and localization data needs to be supplied, then dimension and the signature of the building is set.

6.6 BLESS Services

For testing functionality of the prototype of BLESS, five BLESS services were implemented. Further, five services which are meant only for debugging were created. Former services are presented in sections 6.6.2 and following. The following section introduces the service skeleton which was created to facilitate development of BLESS services. Further details can be found in [Zil15].

6.6.1 Service Skeleton

The base class of the service skeleton library is `ServiceSkeleton` which each BLESS service implementation should extend. Its features are presented in the following.

Publish Services for Discovery

When `ServiceSkeleton` is instantiated it announces the service in the local network. This can either be done as UPnP entity using SSDP or via mDNS. Again the same libraries were used as for the implementation of MultiApp as described in section 6.2.1.

Definition of Remote Interface on Server Side

For defining remotely invocable interfaces the service skeleton library defines Java interface `RemoteInterface` which must be inherited by every actual remote interface defined by a BLESS service. Consequently the according implementations must implemented `RemoteInterface`. This is illustrated as UML class diagram in figure 6.4 using the example of door guard service from section 6.6.4. Note that `RemoteInterface` does not define any methods. It is merely used for marking its implementations as remote interface.

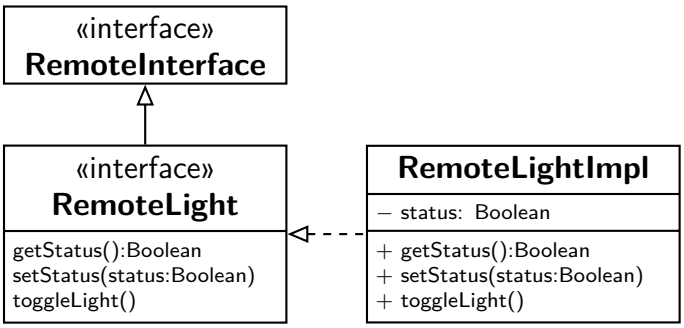


Figure 6.4: UML class diagram of interface for door guard service

Accessing Remote Interface on Client Side

Methods defined by a Java remote interface – e.g. see listing 6.1 – by the SP need to be made accessible as JavaScript calls by the CP. For this purpose the client side requires corresponding client stubs. These client stubs are also created by the service skeleton when `ServiceSkeleton` is instantiated.

```
1 public interface RemoteDoor extends RemoteInterface {
2     /**
3      * Get status of guarded door.
4      * @return true if door is open, else false
5      */
6     boolean getDoorStatus();
7 }
```

Listing 6.1: Definition of remote interface for door guard service

Listing 6.2 shows the generated JavaScript code for the door guard service. It includes methods comments provided in the Java interface as well as the type of arguments and return values. This is necessary because JavaScript uses no type declarations. However, they provide valuable information to the developer of services.

```
1  var RemoteCall_RemoteDoor = {
2
3      /**
4       * Get status of guarded door.
5       *
6       * @return true if door is open, else false
7       *  (Java type: boolean)
8       */
9      getDoorStatus : function(){}
10 };
```

Listing 6.2: Generatated JavaScript client stub for door guard service

At this time all methods are empty shells whose content is dynamically added at runtime. This is possible because JavaScript is a dynamic language which allows to modify functions during runtime. The CP invokes remote calls as shown in listing 6.3. Callbacks are handled using the JavaScript concept of promises, it allows either function `successCallback` or function `errorCallback` to be executed after `setStatus` has completed depending on its success value.

```
1  RemoteCall.setStatus(true).then(successCallback, errorCallback);
```

Listing 6.3: Invocation of remove call including callbacks

Implementation of Remote Call on Client Side

The actual creation of the content of the client stubs is performed on starting the CP, using the function call shown in figure 6.4. Note that the IP address of the server depends on the device which hosts the service. It thus

must be determined by MultiApp using the discovery protocol for BLESS services.

```
1 JSRC.initializeClientStub(RemoteCall_RemoteDoor, <server IP>);
```

Listing 6.4: Call to initialized client stub at runtime

After the client stub is created it performs the serialization of parameters and the marshaling of the function call itself. All relevant data is stored in a JavaScript object according to the JSON-RPC format. For sending the request to the service provider, the service skeleton offers two communication modules. The default methods relies on WebSockets which allows reusing a single connection for multiple requests and also allows the server to send messages as long as the connection persists. As fallback – in case the client does not support WebSockets – standard HTTP is used.

Implementation of Remote Call on Server Side

The server side answering of remote calls is handled by Java class `JsRcCore` which is also provided by the service skeleton. It embeds the web server Jetty[1] which is used to host entry points `/http` and `/websocket` for the two communication methods. During initialization of `JsRcCore` the definition as well as the implementation of the remote interface must be provided. This allows the service skeleton to initialize the service specific remote interfaces as well as handling incoming calls appropriately. This includes deserializing method argument from the JSON-RPC notation into a Java object, invoking the given implementations of the remote calls, and serializing the return values including sending them to the client. Also for executing methods without return values an answer is sent to the client so that it learns that the remote call was successful. In case of a failure a JSON-RPC defined error object is returned containing the cause of the error.

The service skeleton also contains a pusher control object. It manages registered clients and allows the server part to send messages via the pusher to the registered clients. The pusher is detected autonomously according to the discovery protocol.

[1] http://eclipse.org/jetty

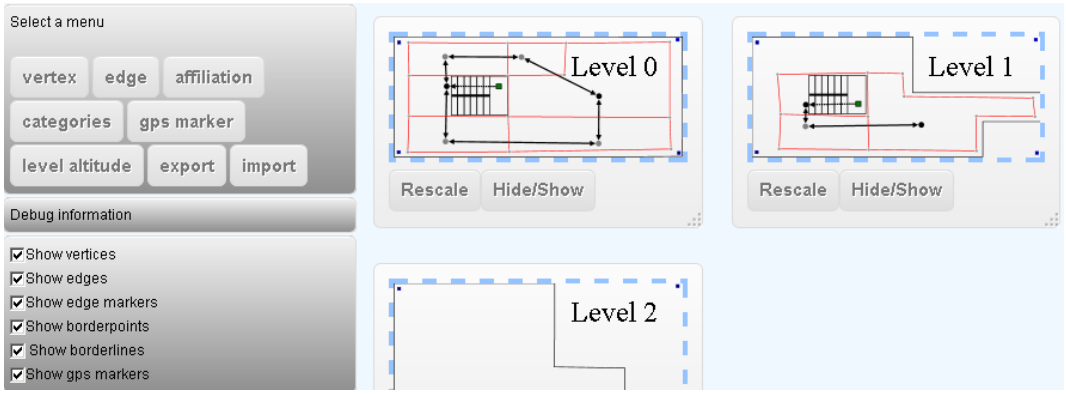


Figure 6.5: Overview of Editor's View of SvgNaviMap

6.6.2 Navigation

The Navigation service described in section 3.4.1 was implemented. Its SP is a simple BLESS service relying on the service skeleton. It does not provide a remote interface and is thus implemented as offline service. The CP is a web-based map viewer including navigation functionality which uses data provided by any BLESS building. It is based on the User's View of SvgNaviMap [3].

Creating Maps with Navigation Data

Maps with navigation data need to be created using Editor's View of SvgNaviMap as explained in [3]:

> [First, a new SvgNaviMap project has to be created and the required building maps need to be added as SVG files.] Next, routing information has to be created for the SVG using the Editor's View (figure 6.5). Using a menu, all routing information elements described above can be created. If available, routing information can also be imported.
>
> First, vertices need to be added. In general, in the center of each room a POI vertex should be placed. Additionally, a room label may be added as description. Helper vertices close to each door will assure that direction arrows will not pass through walls. Next, edges are added to connect vertices. For edges

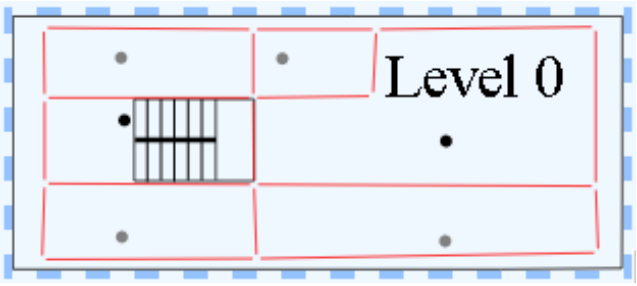


Figure 6.6: SvgNaviMap floor map showing routing vertices and affiliation areas

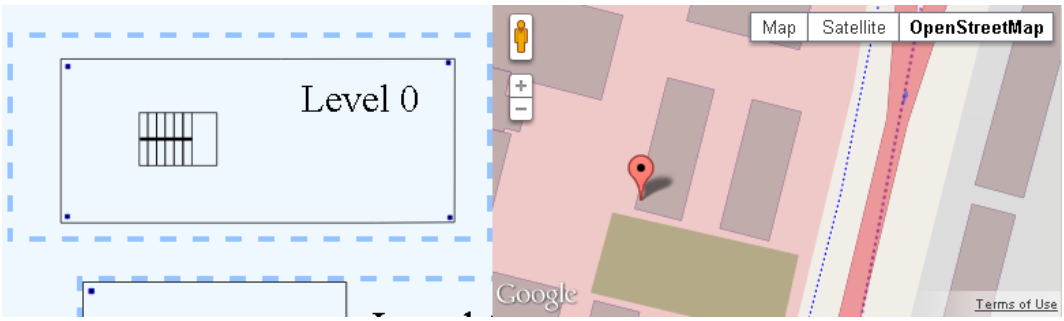


Figure 6.7: SvgNaviMap floor map showing GPSmarkers on the left and an according anchor on an OSM map on the right.

connecting different levels, stepmarkers are inserted automatically. Further, affiliation areas need to be assigned. All vertices need to be inside their corresponding areas, as shown in figure 6.6.

GPS Markers are first added to the SVG map. Afterwards, their corresponding GPS position is set either by explicitly entering latitude and longitude or by choosing a position on a map as shown in figure 6.7. Finally, lower and upper heights of all levels need to be set.

User's View of SvgNaviMap / BLESS Navigation

Once building's map and navigation data is created, it needs to be made available to user's by adding it to a BLESS building. Even though as soon as MultiApp installs the building, its data is available to all BLESS services, it is required primarily by the navigation service.

The navigation service provides mainly two functionalities: It displays the current position which is obtained from MultiApp and it allows to calculate and display shortest routes between two points. Internally, an inverted Dijkstra algorithm is used which calculates routes from any source node to the destination. This way, the shown navigation directions can be updated whenever the own position changes without recalculating the route. In case a route contains floor changes, the navigation service shows stepmakers. They indicate where floors are entered and exited.

Further, an interface is offered which allows CPs of other services to display POIs and routes.

6.6.3 Automated Door Bell

The automated door bell service was implemented as detailed in section 3.4.2. The door bell is accessible by the service provider such that it could be activated by the CP of a service.

When started, the service registers a large geofence of about 1 km diameter around the bell. When the geofence is entered, the services registers for online events. As soon as the SP of the services is available, the bell is rung. Before the bell is rung again, the geofence has to be left. The service is supposed to be running in background at all times. It is implemented as offline service, so it can be started even if the SP is not available.

6.6.4 Door Guard (Home Monitor)

The home monitor from section 3.4.9 was implemented as door guard. The door is equipped with a state sensor which can be read by the service provider and thus the SP of the service. When the CP of the offline service is started, it registers for push messages with the SP as soon as it becomes available. The SP now pushes every state change of the door to each client. The status is received silently by the CP and stored persistently. Additionally, the CP registers a geofence which is triggered as soon as the building is left. The CP now checks the last known state of the door, if it was open an acoustic and vibration alarm is raised.

6.6.5 Heating Status (Information Request)

The heating status service is a simple information request service. When it is opened by the user, the current hot water supply temperature as well as the state of the circulation pumps of the heating system is displayed. Status data can either be obtained via remote call from the SP or directly from the heating server using a HTTP GET request. When the SP is available the former approach is chosen, otherwise the later. If the called URL is globally available via Internet, the heating status can even be obtained when not being inside the service's local network.

6.6.6 Light Switch (Switch Service)

The SP of the light switch service offers a single function for toggling the light state. When the user launches the CP the light is toggled and the CP exits. The service is intended to be used only when the SP is available (online service) and when the user is within the room in which the light is controlled. Feedback about the light state from the service is thus not required, but obtained directly by the user. The implementation of the light switch includes a basic authentication and authorization method: It holds a list of allowed users and relies on the e-mail address contained in light toggle requests (cf. section 5.5.1) to identify the user. The toggle operation is only performed is the requesting user is listed a allowed user.

6.7 Lines Of Code

To provide an estimate about the size of the BLESS framework and about the development efforts, this section provides the numbers of lines of code for the different components.

6.7.1 MultiApp

Including GUI which is defined in Android as XML files, MultiApp counts approximately 20,000 lines of code. About the same amount of lines are covered by the used UPnP library Cling. It includes about 1000 lines of XML

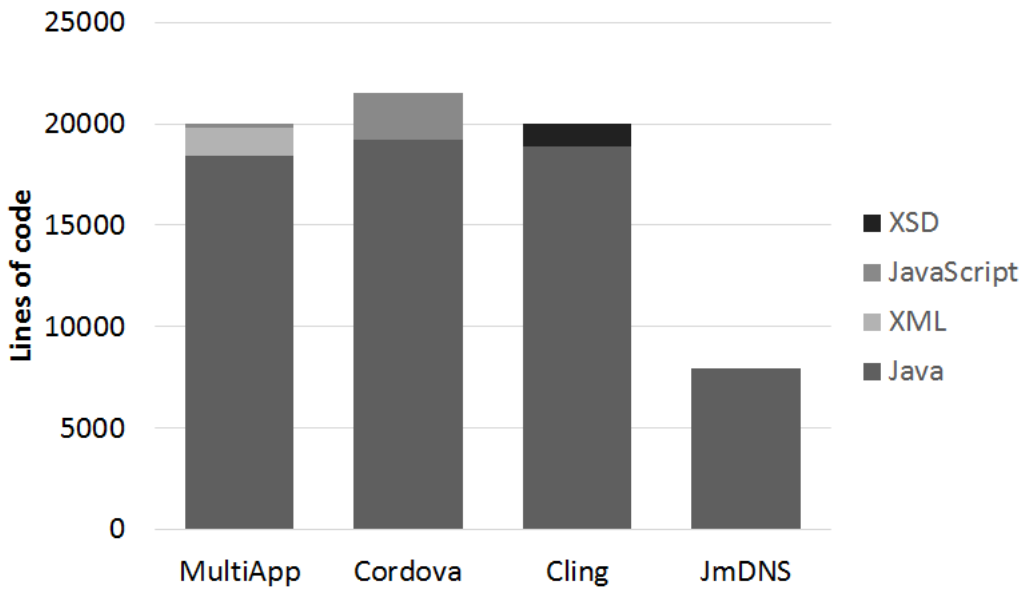


Figure 6.8: Overview of lines of code of MultiApp and its libraries

schema data (XSD) describing the format of UPnP service and device description files. For the later approach using DNS discovery instead of UPnP, the library JmDNS with almost 8000 lines of code is used. The library Cordova which is also deployed by MultiApp counts more than 21,000 lines of code. A graphical representation of these number is shown in figure 6.8.

6.7.2 Services

Except for the navigation service all services are very simple ones. Including helper functions but exempting libraries, the JavaScript code of each CP counts around 200 lines of code, the Java code of each SP between 150 and 300 lines of code. Including HTML and CSS code for the GUI, each service consists of less than 800 lines. More details are shown in figure 6.9. The navigation service contains a similar amount of CSS, HTML, and Java code, however, it counts around 2500 lines of JavaScript code. This is due to the logic which reads and parses the navigation data, performs the shortest path algorithm, and dynamically builds the navigation overlay.

As comparison, a native Android app – which registers with Google's push service (GCM), sends its device ID to a custom server, and displays received push messages – counts about 2000 lines of Java code. Addition-

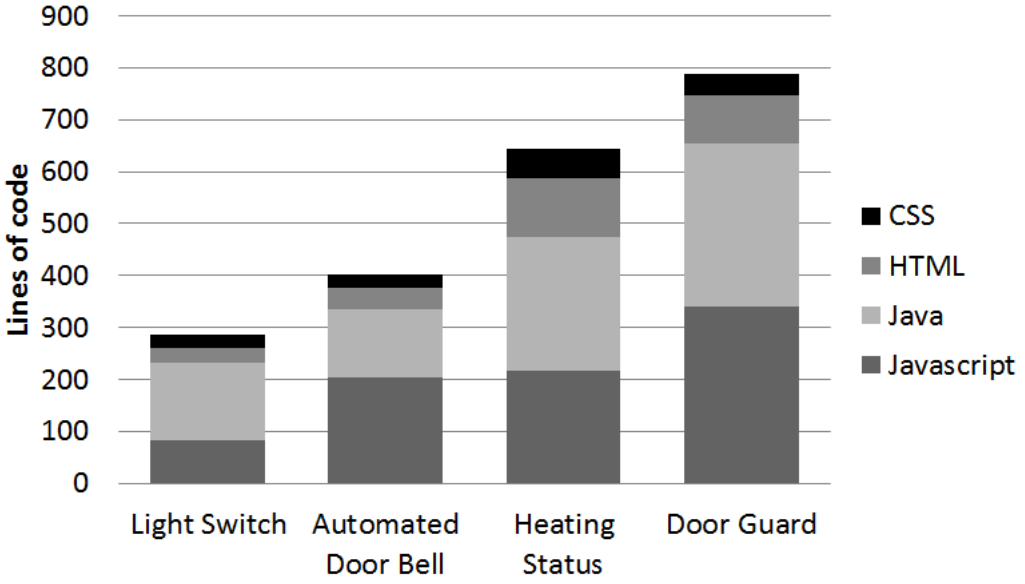


Figure 6.9: Overview of lines of code of implemented BLESS services

ally, about 200 lines for the GUI are required and another 100 lines of PHP code for the server. Further, the developer must have registered the app with Google to obtain an API key required for using GCM.

Keeping the required Java code as low as 300 lines of code for BLESS services is only possible because the functions for networking discovery as well as communication are part of the service skeleton. The m/DNS version counts about 1800 lines of code, while the UPnP version requires about 2400 lines. As comparison figure 6.10 shows these values together with the navigation service.

6.7.3 Buildings

Also BLESS buildings are network discoverable entities and are thus also built on the service skeleton. As a result buildings discoverable via m/DNS only count 56 lines of code. The UPnP versions require about 250 lines. However, the larger parts of a building are the SVG map, the navigation data, as well as the databases for fingerprinting and WiFi DB localization. These value are graphically represented in figure

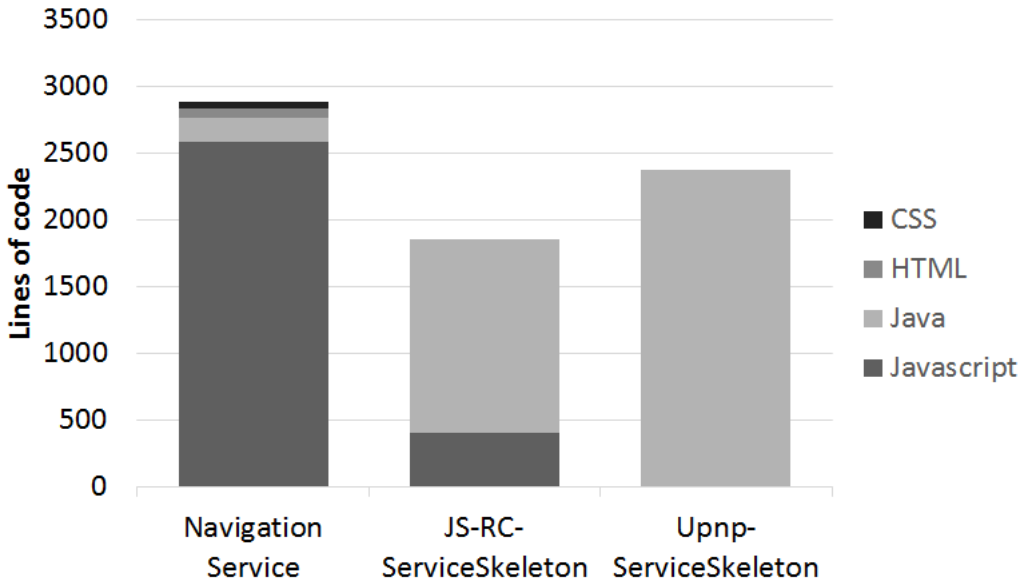


Figure 6.10: Overview of lines of code of service skeletons compared to navigation service

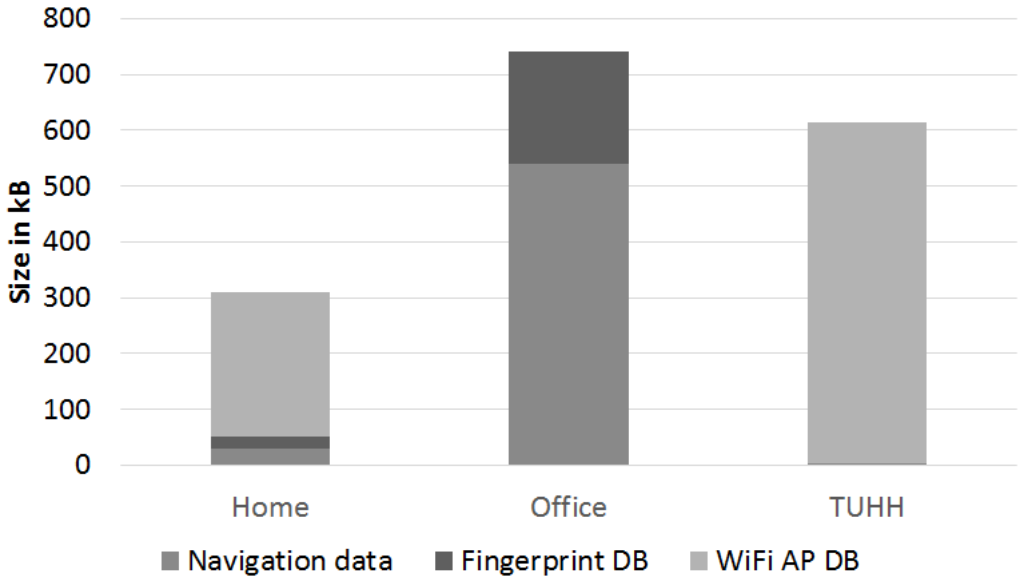


Figure 6.11: Size of navigation data and localization databases for the three sample buildings

Chapter 7

Applying BLESS to private house hold

This chapter presents the setup (sections 7.1 and 7.2), procedure (section 7.3), and results (section 7.4) of a case study demonstrating the feasibility of the BLESS concept. The goal is to provide subjective opinions and experiences of test users. No measurements are performed nor is test data collected as this data would primarily provide an indication about the quality of the implementation of BLESS. However, the goal is to provide data for an evaluation of the concept which was presented in chapter 5. The evaluation itself can be found in chapter 8.

During development and testing more similarities between BLESS and openHAB were encountered as originally expected. The second part of this chapter – starting with section 7.5 – studies differences and common features of BLESS and openHAB. The evaluation assesses how both projects may be combined to create added value.

7.1 Description of Field Test Environment

The field test was performed in a three-story, duplex house. In one part walls and floors are made of concrete, the other is mainly a wooden, lightweight construction. The total floor area is about $200\,m^2$. The house is equipped with two WiFi access points (APs).

In the house an additional bell was installed and the backdoor was

equipped with a sensor able to determine the opening state of the door. Further, a floor lamp in the living room was connected to a digitally switchable power plug. The mentioned devices are controlled wirelessly using the proprietary FS20 protocol[1]. According radio signals can be emitted and received using a dedicated network-attached device (FS20Manager[2]) which was installed close to the three wireless devices.

All BLESS entities were executed on a Linux computer located in the basement. This BLESS provider has full access to the FS20Manager. All relevant FS20 signals are forwarded to the provider computer via cabled network.

7.2 Preparation for the Field Test

The preparation for the field test started with installing an additional WiFi AP in the basement of the building due to a very poor signal strength in that area. Afterwards collecting WiFi fingerprints for localization began. At 26 different locations both inside the house as well as in the close proximity around the house fingerprints were taken. In total 64 different APs were detected. At each location at least two and at the maximum 21 APs were visible.

Afterwards collected WiFi data was processed to be compatible with the machine learning algorithms presented in section 6.2.9. Additionally, data for WiFi DB localization was queried from the Wigle project's website[3] for an area of about $100\,\mathrm{km}^2$. Further, in the vicinity of about $500\,\mathrm{m}$ around the house data was collected in person. Both data sets were combined and transformed into the required format. It includes 10569 positions of WiFi access points.

Finally, all BLESS services described in section 6.6 were activated on the Linux computer. Also the pusher from section 6.4 and BLESS building Home from section 6.5 were executed.

[1] http://www.elv.de/fs20-funkschaltsystem.html
[2] https://web.archive.org/web/20150102193202/http://www.crazy-hardware.de/
[3] https://wigle.net/

7.3 Procedure of Field Test

The field test was conducted from 21 to 31 March 2015. On the first day all devices were activated, all services were started and MultiApp was installed on the smartphones of the two test persons living in the house – called A and B in the following. The provider and its services stayed active for the whole duration of the test. MultiApp was supposed to be running at all times on the mobile test devices. However, it could be started and stopped at discretion of the test persons. As discovery protocol m/DNS was used as described in section 6.2.1.

At the beginning of the test both test persons launched MultiApp. All offered BLESS services were detected instantaneously as well as the BLESS building. All these entities were installed on both test devices. As user A was mainly at home during the duration of the test, the automated door bell service was only installed on test device B. It stayed active during the whole field test. The door guard service was enabled and stayed active on both test devices. The remaining three services were opened and closed as required by the test users. The navigation service – not needed by inhabitants in their own home – was only used occasionally by user B in order to visualize the currently detected position.

7.4 Results of Field Test

Programmatically all BLESS entities as well as MultiApp worked well. No crashes nor inexplicable program behavior was observed. Initial detection and installation of entities within MultiApp worked straight-forward and smoothly. Also communication between entities and MultiApp functioned well, including push messaging via the BLESS pusher.

However, serious issues resulted from the applied localization algorithms. First, it often returned inaccurate localization data on room-level. The navigation service was thus unusable because the current location often jumped from one end of the house to the other. The door guard service often issued warnings even though the users did not leave the house.

Second, geofence events were not issued quickly enough. Thus, the door

bell service never rang before user B entered the house, sometimes it took several minutes until the bell was rung. This might be caused by too large scan intervals in combination with a delayed service detection. However, the third problem – a very high battery consumption – is the result of too many localization scans. For device B, instead of lasting 3-4 days, the battery was gone after a little over a day. For device A the battery depleted in even less than a day with MultiApp running.

It has to be noted though, that there was a big quality difference between the localization algorithms. The BSSID-based localization algorithm reliably reported connecting and disconnecting to WiFi access points mapped to known buildings. However, even using the optimized WiFi detection offered by the operating system introduced a latency between multiple seconds up to several minutes. Additionally, the implementation of the BSSID-based localization algorithm waits until the connection to the WiFi is established instead of issuing enter notifications as soon the network was seen. This causes an additional delay of usually a few seconds.

The other algorithms – namely the machine learning techniques as well as the WiFi DB localization – could not rely on an existing scheduler for starting locations scans. Instead dynamic scan interval described in section 6.2.10 were used. Considering slow geofence detection, the intervals were too large, however, taking the high energy consumption into account scans should be started less frequently.

During the test it was also observed that the detection of entities via DNS discovery was not reliable on entering a building. Often only a fraction of available services were detected. However, since the detection of the current building worked well using the BSSID-based localization technique and all linked services were considered to be online implicitly (cf. section 5.9.1), this issue had no impact on functionality of BLESS during the field test.

Additionally, the wireless connections to external devices posed an issue due to unreliability. Sometimes the light switch service did not switch the light without any error message. Also some opening state notifications of the backdoor never reached the door guard server and the door bell did not ring even if the ring command was issued according to the log files. All

three phenomena are most likely to occur due to lost FS20 packets but are not caused by a design failure of BLESS.

The performance of MultiApp was acceptable. Launching MultiApp itself as well as installed services takes similarly long as starting other native apps. However, the time until a service is completely loaded highly depends on the JavaScript implementation. For example, while the heating status is displayed almost instantly, it takes several seconds until the map is shown by the navigation service.

Another issue which makes MultiApp rather user-unfriendly is the fact that starting services involves opening a full screen WebView element (cf. section 6.2.5). This implies that when MultiApp needs to pass some kind of data, e.g. a geofence event, to a CP of a service, then the service overlays the currently active app. Even though all services were designed such that they terminate themselves after they were activated by MultiApp for accepting data only, a brief pop-up is still annoying and is interrupting the user's workflow.

7.5 Comparing with openHAB

During development of MultiApp and testing of openHAB is was determined that by introducing small modifications to implemented location-linked services they can be converted into smart home applications being executable by openHAB. In order to compare both projects the four services Automated Door Bell, Door Guard, Heating Status, and Light Switch (sections 6.6.3 throughout 6.6.6) were realized – with some modifications – using openHAB.

For this, it was necessary to allow openHAB to determine the presence of the Android devices. It was further required to circumvent client-side logic and actions. Details about these modifications are described in the following sections.

7.5.1 Presence Detection with openHAB

openHAB provides a module called *NetworkHealth*. It can be configured to periodically connect to network devices and to report back their availability. A client which responds is at home. A timeout signals absence. For this test the two test devices A and B were checked every 5 min. The detected presence status of the devices are stored as openHAB items and are thus available to all smart home applications which are being executed by openHAB.

Note that these modifications contradict the design requirement of avoiding omniscient entities {N2}. Further, since no room-based position is determined, neither the localization requirement {F10} can be considered fulfilled. A complete comparison about which system fulfills which requirements is provided in the evaluation chapter in figures 8.1 and 8.2.

7.5.2 Realizing Test Services using openHAB

Implementing the four test services with openHAB and thus converting them from location-linked services to smart home applications required some modifications.

Without client-sided logic the automated door bell service (section 6.6.3) had to rely on openHAB to detect its presence. The openHAB implementation of this application thus monitors the item indicating the presence of test device B. When it changes from absent to presence, the bell is rung.

The openHAB implementation of the door guard service from section 6.6.4 cannot store the latest door state on the smartphone. Neither can the client-side detect when the user leaves the house, nor can it execute actions on the smartphones to inform the user about the door being unlocked. Instead the door guard application relies on the openHAB server to detect the presence of users A and B. Also the state of the door is stored within openHAB as item. When the absence of both test devices is detected and the door sensor indicates that the door is open, the openHAB application issues a command which causes the shutters to close.

Neither the design of the heating status service from section 6.6.5 nor of the light switch service from section 6.6.6 need to be changed, as both

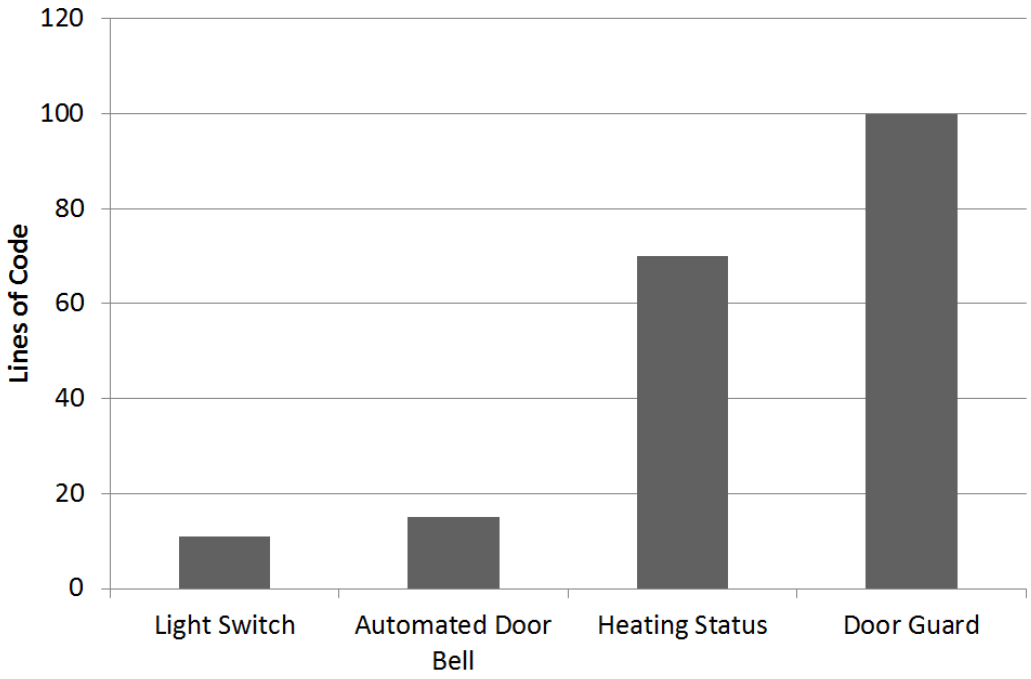


Figure 7.1: Approximate lines of code of openHAB test services

services are actually already smart home applications. In case of the heating status application the client is only used for displaying data to the user, for the switch service the smartphone is only used for sending a switch input to the server.

The navigation service could not be realized as openHAB application due to its complex client-side logic and GUI.

Applying presented simplifications, implementing the four test services required very little work. The approximate amount of number of lines using openHAB's configuration language for the four test services in shown in figure 7.1. The numbers include both, lines for configuring openHAB items as well as lines for implementing application logic using openHAB's Java-based configuration language. It does not include code for presence detection of the test devices.

7.5.3 Procedure of openHAB Test

The field test was conducted in June 2015. The openHAB server was executed during the whole time on the same computer as previously the BLESS

entities. All previously described applications were activated and the presence detection was configured and enabled for devices A and B.

During the test phase both participants launched and closed the openHAB app on their smartphones as required. The via browser accessible web-based interface was not used. The door bell and the door guards applications could not be stopped by the test persons. The heating status and light switch applications were started and used at the test persons' discretion.

7.5.4 Results of openHAB Test

Programmatically the openHAB server software, the openHAB Android app, as well as the implemented services worked well. No crashes nor inexplicable program behavior was observed.

Again, the major problem was the presence detection. As it was triggered only every five minutes, the automated door bell never rang before user B entered the house. Often test devices disappeared without obvious reasons, resulting in shutters going down even though at least one test person was present. After re-detecting test device B a false alarm ringing was the result from the automated door bell service.

Also the wireless FS20 signals were reported to be unreliable. At times the bell was not rung even though the presence of device B changed from absent to present.

As there was no need to keep the openHAB app running, it was often closed by the test persons. This resulted in a cumbersome light switching, since first the openHAB app had to be started. It took up to 3 s for the openHABclient to find its server and to list available applications. Afterwards the light application had to be invoked.

Due to unreliable presence detection it was very disadvantageous that the test users could neither disable the door bell nor the door guard application. It was positively perceived that the presence detection did not noticeably affect the battery consumption. Note that during pre-tests the detection interval was set from 10 s to 30 s which caused a high energy drain.

Chapter 8

Evaluation and Conclusion

This evaluation chapter first analyses the fulfillment of requirements from chapter 4 in section 8.1. Section 8.2 completes this chapter with a conclusion about this dissertation.

8.1 Fulfillment of Requirements

In chapter 4 the innovative service provisioning system BLESS was envisioned and 9 non-functional requirements as well 31 functional requirements were identified and described. This section evaluates which of these requirements were fulfilled by the design from chapter 5 as well as the implementation from chapter 6. It discusses reasons for unsatisfied items.

8.1.1 Non-Functional Requirements, BLESS

An overview of the fulfillment of the non-functional requirements is provided by table 8.1. As comparison it includes how many non-functional requirements the smart home system openHAB fulfills (cf. section 8.1.2).

No Single Point of Failure {N1}

The design of BLESS allows each entity to be hosted by a different provider. Each entity is a stand-alone Java program and can thus function independently. Also the discovery protocol as defined in section 5.9.3 does not depend on a service directory which would in case of failure render the

whole system unusable. However, the design of BLESS introduces multiple interconnections between entities. While no failure of a single entity can break the whole system, several components may have a negative impact.

During development and testing it became obvious that two entities are particularly important for a smooth operation of a BLESS system. It is the pusher and building together with sub-building entities. If the former fails, services cannot send push messages to its CPs. While this is a severe limitation, it does not break the whole system and does thus not violate requirement {N1}. Even more severe is the failure of the building entities for users entering a never-visited building. In this case detected services cannot be linked to the building. Filtering by building becomes impossible. Also localization will fail as no localization data is available. Again, while these issues are severe, only parts of BLESS are affected and only for new users. Requirement {N1} is not violated, either.

The effect a failed service has on other BLESS entities is minimal. Only other services, that use function {F30} to invoke the failed service and only if the failed service requires its SP, are affected. This is no single point of failure, either. Further, the implementation from chapter 6 follows the design and thus also satisfies requirement {N1}.

Other circumstances in which the BLESS system may fail are conceivable, nonetheless. Examples are: The network fails and MultiApp cannot communicate with any BLESS entity; all entities are executed by the same provider and that computer fails; or the MultiApp instances of all users crash due to an illicit configuration of the BLESS system. However, all these failures are not due to a poor design. The last failure is theoretical. No evidence of a problem of this type was encountered during the field test.

No Omniscient Entities {N2}

The design of BLESS satisfies the requirement of avoiding all types of omniscient entities closely. In particular, there is no storage for data of multiple buildings, no directory listing all service or building entities, and no unit which is aware of all users or their permissions or locations. This design was implemented accordingly. For design and implementation requirement {N2} is thus fulfilled.

Building Data Provided by Building {N3}

Requirement {N3} demands that building data is to be provided by the building to the clients. This was realized in design and implementation by introducing building entities whose main purpose is to provide this data. They also provide a name for the building as well as the geographical location and dimensions. The requirement is thus fulfilled twice.

Local Network Only {N4}

The BLESS system was designed in such a way that it can work completely independent from any Internet servers. The implementation of the prototype follows this design. Both parts thus fully satisfy requirement {N4}.

Secure Connections {N5}

The design of BLESS suggests to use certificates for encryption. As details are missing, however, this requirement {N5} is fulfilled only partly. The implementation does not provide any support for secured connections.

Services as Powerful as Apps {N6}

Even though a web-based application can never be as powerful as a native app, the design of BLESS allows to make any additional functionalities available to BLESS services. For the design, requirement {N6} is considered fulfilled. The implementation supports all functionalities required by the sample services from section 3.4 by relying on Cordova and HTML5 as well as by explicit implementation by MultiApp (cf. section 6.2.5). Even though in particular Cordova offers many more interfaces, BLESS services cannot be considered to be as powerful as native apps. In order to make additional functionalities available to BLESS services, extending, recompiling and reinstalling MultiApp is necessary. For the implementation requirement {N6} is considered to be fulfilled only partly.

Ease of Development and Administration {N7}

For implementing BLESS services web technologies like JavaScript, HTML5, and CSS are used. As helper library the widely used Cordova project is incorporated. Remote server invocations rely on JSON encoding and adhere to the JSON-RPC packet structure. For security in particular authentication and encryption the design of BLESS envisions the use of certificates using OpenPGP standards. For displaying building maps, positions therein and navigation routes, the vector graphics format SVG is used. Next to these well-known state of the art technologies, also well-known and well-documented algorithms were used, e.g.: Shortest routes are calculated using the Dijkstra algorithm; room-based localization was implemented relying on existing machine learning algorithms Random Forest and Hyperbolic Location Fingerprinting. Since all these technologies can be considered standards, no additional effort is required by the developer to learn, e.g., a new programming language. This is in accordance with the requirement of facilitating development efforts {N7}.

Further, the service skeleton presented in section 6.6.1 helps the developer to create BLESS services by handling remote invocations including server communication and marshaling. It also facilitates server push messages.

Adding a new service just requires launching a Java program. No configuration except for the service itself is necessary. Hence, requirement {N7} is considered to be fulfilled both for design and implementation.

Platform Independence {N8}

The SP of a service is programmed in Java and is thus executable in Java runtime environments which are available for every major, current operating system. The CP is implemented with web technologies by design runnable within MultiApp with any mobile operating system. However, as MultiApp was implemented only for the Android platform, requirement {N8} is fulfilled only partly with respect to implementation.

Ease of Usage {N9}

The field test as shown that the implementation MultiApp is easily usable. It is a regular Android app offering a user interface similarly to other apps. By design, requirement {N9} is thus considered to be fulfilled. However, considering that the implementation does not include certificate handling with authorization and encryption, requirement fulfillment is not completed.

8.1.2 Non-Functional Requirements, openHAB

The *Home Automation Bus*, which is the origin of the name openHAB, collects all available data – including the presence state of users – and makes its accessible for all items connected to the bus. It must thus be considered an omniscient entity and single point of failure. Requirements {N1} and {N2} are thus violated.

Building data is not supported by openHAB. Requirement {N3} is thus not fulfilled. In the other hand, requirement {N4} is fulfilled as openHAB does not require any Internet connection.

Requirement {N5} of authenticated and encrypted communication is fulfilled partly. The openHAB system can be configured that the client is required to authenticate by sending user name and password before using the system. However, it is not possible for users to authenticate to individual services. Similarly, authentication of the openHAB system can be provided by relying on SSL certificates to secure openHAB's HTTP-based communication, however this is not possible for individual services. Finally, encrypted communication is given by using HTTP over SSL.

Since openHAB is a home automation system rather than a location-linked service system, its services are rather applications which can only accept user input and display status information to the user. There is no client part which can be executed on the smartphone. Consequently services are not slightly as powerful as apps, leaving requirement {N6} unsatisfied.

Existing technologies were used, e.g. openHAB is modularized using the Java OSGi standard and relies on HTTP and SSL for communication. However, for creating services a custom rule engine is used. Admittedly, it is very easy to learn but it does not allow to create complex services either.

Hence, requirement of ease of development {N7} is considered to be fulfilled on partly.

Platform independence on the server side in achieved by using Java. As the client side is a simple web application, it is also platform independent. However, as offered functionalities are severely limited, requirement {N8} can only be considered to be fulfilled partly. Similarly, the requirement of ease of usage {N9} is satisfied only partly. Services are easily usable via offered apps or web browser, however, the more services openHAB offers the more complicated handling gets since there is no service filtering by location. Neither is there filtering by user; all users can access all available services. With these simplifications in mind, requirement ease of usage {N9} can only be considered to be fulfilled partly.

Non-functional requirement	Requirement summary	BLESS Design	BLESS Implementation	openHAB
{N1}	No single point of failure	✓	✓	✗
{N2}	No omniscient entities	✓	✓	✗
{N3}	Building data provided by building	✓	✓	✗
{N4}	Local network only	✓	✓	✓
{N5}	Secure connections	○	✗	○
{N6}	Services as powerful as apps	✓	○	✗
{N7}	Ease of development and administration	✓	✓	○
{N8}	Platform independence	✓	○	○
{N9}	Ease of usage	✓	○	○

✓: Fulfilled, ○: Partly fulfilled, ✗: Not fulfilled

Table 8.1: Comparison of fulfillment of non-functional requirements from section 4 of BLESS design, its implementation, and of openHAB

8.1.3 Functional Requirements, BLESS

An overview of the fulfillment of the functional requirements is provided by table 8.2. As comparison it includes how many functional requirements the smart home system openHAB fulfills (cf. section 8.1.2).

Detect, Filter, and List Services {F1}, {F2}, {F3}

Usually services were detected automatically during the field test as designed. However, sporadically detection of services failed until MultiApp was restarted. The exact reason could not be determined. However, it is likely that either within the used JmDNS library or the usage thereof caused the problem. It is to be expected that an implementation which follows the much simpler discovery protocol as presented in section 5.9.3 solves the issue. Currently the implementation thus fulfills requirement {F1} only partly.

Similarly, also the filtering of services (requirement {F2}) was implemented only partly. This is due to the lacking implementation of user authentication using certificates. Details are provided in section 6.2.3.

The presentation of available services worked well during the field test, completely fulfilling requirement {F3}.

Service Description {F4}

The design of BLESS envisions the usage of a custom discovery protocol for detecting services. This protocol includes a field for the description of services. The design thus fulfills requirement {F4}. However, the implementation of the prototype tested during the field test uses m/DNS as discovery protocol (cf. section 6.2.1). It only support a friendly service name but no service description. The implementation thus fulfills requirement {F4} only partly.

Trust Level Support {F5}

According to the design described in section 5.5.1, certificates are used to establish trusted relation between users and services. In particular ser-

vices signed with a known and trusted certificate are considered trusted. The design thus fulfills requirement {F5}. However, the implementation of MultiApp handles signature management in a strongly simplified manner (cf. section 6.2.2). Since these applied signatures are not forgery-proof, requirement {F5} is fulfilled only partly.

Service/User Provides Authentication {F6}, {F7}

Section 5.5.1 advocates to use personalized certificates for authentication and outlines how this could be done. As this sketch does not provide sufficient design details in order to implement it, according requirements {F6} and {F7} are considered to be only partly fulfilled. The implementation refrains from using certificates but uses strings of e-mail addresses for user and service authentication. As this allows to identify communication partners but does not provide any security measures, also the implementation of referenced requirements are considered to be fulfilled only partly.

Services Restricted by Permissions {F8}, {F9}

The design of BLESS clearly explains which actions should be restricted by enforcing permissions and how the user should be able to set these permissions (cf. section 5.5.3). This fulfills requirement {F8} for the design part. However, as this specification was not realized for the prototype, the implementation does not satisfy this requirement. Consequently the requirement to notify the user about which permissions are currently being used {F9} is not implemented nor is it included in the design, either. However, no difficulties are to be expected when adding this detail. It is thus considered a pure routine piece of work and was hence omitted in this work.

Localize Users {F10}

According to design BLESS integrates a flexible framework for integrating localization technologies which are executed on the mobile device using pre-loaded localization data from a BLESS building entity. However, the implemented algorithms do not reliably achieve the required room-based accuracy and thus fulfill the requirement only partly.

Offline Support {F11}

According to design specifications from section 5.4.1, a BLESS service should be able to start even while the according SP is unavailable. The prototypical implementation achieves this behavior by loading all required files for the CP upon installation. Both design and implementation thus fulfill requirement {F11}.

Two-Way Communication {F12} and Service Update {F13}

During the field test communication between MultiApp and BLESS servers worked smoothly. The design and implementation goals are thus reached. Same holds for the requirements of allowing to update BLESS services.

Wake Events {F14}–{F19}

In order to (re-)activate stopped and minimized services (cf. section 5.4.2) wake events are envisioned by the design. All required events were implemented within MultiApp as explained in section 6.2.6. According requirements {F14} throughout {F19} are thus fulfilled.

Run in Background {F20}

Making BLESS able to run in background is inevitable in order to activate multiple services at a time. According to design this feature was implemented by the MultiApp prototype (cf. section 6.2.6). Both design and implementation thus fulfill requirement {F20}. However, it has to be noted that the field test revealed a shortcoming of the implementation concerning passing data to a service running in background as detailed in the last paragraph of section 7.4.

Capabilities for CP {F21}–{F31}

The required capabilities for the CP of a service are made available in three different ways: 1) via standard Cordova plugins, 2) via Cordova plugins offered by MultiApp, and 3) via JavaScript objects offered by HTML5. Details

are provided in section 6.2.5. During the field test all functions worked according to design. Both design and implementation thus fulfill requirements {F21} throughout {F31}.

8.1.4 Functional Requirements, openHAB

Software openHAB supports automated discovery of the openHAB server which in turn makes the service available. Detecting individual services themselves is thus not possible and can hence not be listed. Requirement {F1} is therefore fulfilled only partly, requirement {F3} is not fulfilled. Filtering services by user or user position is not supported, requirement {F2} not satisfied. Requirement {F4} demands services to include a description, which is not possible for openHAB. However, above the entry for each service an entry can be added which contains a simple description string. Requirement {F4} can thus be considered fulfilled partly. Trust levels for services are not supported, requirement {F5} not satisfied.

Requirements {F6} and {F7} of mutual authentication are fulfilled partly. The openHAB system can be configured that the client is required to authenticate by sending user name and password before using the system. However, it is not possible for users to authenticate to individual services. Similarly, authentication of the openHAB system can be provided by relying on SSL certificates to secure openHAB's HTTP-based communication, however, this is not possible for individual services.

Since openHAB is a home automation system, it does not support the concept of services being executed on the client-side. Consequently, the concept of permissions is not applicable. Permission requirement {F8} is thus not fulfilled, neither is requirement {F9} concerning notifying the user about permissions in use.

The presence detection implemented for the field test must not be considered a localization technique of openHAB. It is rather networking technology which works well for Android devices. It does not work, e.g., for Windows Phone. Further, it is performed by the server, not the client; and it does not achieve the demanded room-based localization. Localization requirement {F10} is hence not fulfilled.

Also due to the nature of openHAB, for displaying the user interface of a service, direct server communication is necessary. Consequently, requirement {F11} is not satisfied. Further, there can be no push messages from server to client; only communication initiated by the smartphone is possible. As a result requirement {F12} is fulfilled only partly. Additionally, there is no CP which can be updated or offer functionalities, nor are there background services which can be woken. Requirements {F13} through {F31} are thus not fulfilled.

8.2 Conclusion

Smartphones are constantly growing more popular and become more powerful with each new release. In Germany smartphone penetration rose in January 2016 to 74% [eMa16]. The smartphone is for many people a permanent companion, e.g. for staying connected with friends and business partners, for being reminded of upcoming calendar events, and for staying informed about happenings in the world. It has thus become the personal digital assistant which the PDAs during the 1990' aimed to be.

At the same time intelligent buildings with integrated electronic devices and controls are being constructed. Existing houses are enhanced accordingly. For the future it is to be expected that even more technology will be integrated into all kind of buildings.

All these integrated smart devices need to be controlled and managed. Using the smartphone to control and to communicate with the building systems seems to be the logical choice. In fact as of August 2016, searching for "smart home" in Google Play Store returns about 250 findings. Most listed apps are offered by companies to control their proprietary smart home devices. It thus becomes difficult to find that particular app which is able to control the devices in the building where the user currently is in. Indeed finding the right app at the right time at the right place is getting more difficult because app stores - the only officially supported method of installing new apps on smartphones - are growing rapidly. An increase of 50% of apps in one year is not uncommon for Google Play Store and Apple's App Store, which each offer well more than one million different apps.

Functional requirement	Requirement summary	Design	Implem. BLESS	open-HAB
{F1}	Detect services	✓	○	○
{F2}	Filter services	✓	○	✗
{F3}	List services	✓	✓	✗
{F4}	Service description	✓	○	○
{F5}	Trust level support	✓	○	✗
{F6}	Authenticate services	○	○	○
{F7}	Authenticate users	○	○	○
{F8}	Service permissions	✓	✗	✗
{F9}	Notify permission usage	✗	✗	✗
{F10}	Localize users	✓	○	✗
{F11}	Offline support	✓	✓	✗
{F12}	Two-way communication	✓	✓	○
{F13}	Updating services	✓	✓	✗
{F14}	Wake: User launch	✓	✓	✗
{F15}	Wake: By service	✓	✓	✗
{F16}	Wake: Geofence	✓	✓	✗
{F17}	Wake: Push message	✓	✓	✗
{F18}	Wake: Alarm	✓	✓	✗
{F19}	Wake: SP online	✓	✓	✗
{F20}	Run in background	✓	✓	✗
{F21}	CP: Flexible GUI	✓	✓	✗
{F22}	CP: Audio output	✓	✓	✗
{F23}	CP: User Notification	✓	✓	✗
{F24}	CP: Use accelerometer	✓	✓	✗
{F25}	CP: Use compass	✓	✓	✗
{F26}	CP: Access map	✓	✓	✗
{F27}	CP: Access position	✓	✓	✗
{F28}	CP: Client to server	✓	✓	✗
{F29}	CP: Server to client	✓	✓	✗
{F30}	CP: Invoke other services	✓	✓	✗
{F31}	CP: Private storage	✓	✓	✗

✓: Fulfilled, ○: Partly fulfilled, ✗: Not fulfilled

Table 8.2: Comparison of fulfillment of functional requirements from section 4 of BLESS design, its implementation, and of openHAB

Despite the enormous size of app stores for some areas of applications there is hardly diversity. For example, the Facebook and WhatsApp Messenger apps are spread very wide. Resulting in millions and billions of users trapped in a single provider's services and technologies. This trend can be compared with the automobile and fashion industries, which managed with the beginning of the 20th century to introduce mass-production with standard models and sizes as well as predetermined colors. However, people demanded custom configuration; with the result that nowadays, even individualized customization appears to be within reach [SP13, GMR14].

Also for software products a new approach is required. Many users no longer want to use pre-defined solutions and store their data on servers of the big players as can be seen from the increasing popularity of personalized storing solutions like Synology devices or the ownCloud platform.

In particular when considering that smart home systems may control security relevant devices like shutters and alarms, it is of paramount importance to be able to trust these systems. However, as long as relying on globally accessible Internet servers for obtaining the control app or even for accessing the smart devices at home, some vulnerability may be found at any time by the sheer number of potential attackers worldwide. It is thus justifiably to avoid Internet connectivity for accessing private, security relevant data and devices.

8.2.1 Contributions

This dissertation presents the concept of location-linked services (LLSs). These are distributed applications which are intended to be used in local networks only. Considering that current buildings – even large structure such as university campuses – are equipped with pervasive WiFi coverage, there is often no need for Internet connectivity. A definition for LLSs is presented and reasons for using them are motivated. Next to being used for controlling smart homes also other areas of applications presented.

The design of system *BLESS* for providing LLSs is presented. One of its specialties compared to smart home systems is the capability of executing program logic on the client side. Consequently the client part of a service

is programmed rather than configured. This allows to create very powerful services which are able to perform complex operations and to display customized user interfaces. At the same time it is not necessary to install new smartphone apps for each new service. Instead services are detected autonomously by the BLESS client called MultiApp.

Different from app stores, before installing a LLS it must authenticate itself. Also user authentication is envisioned. The design of both processes relies on public/private certificates which are also used for establishing encrypted connections. For users this implies that no passwords and user names need to be remembered. Instead when signing up for authorized services in a building, users need only to prove once that they are the owner of their certificate. For creating new LLSs tools and libraries are presented which facilitate development and administration. By providing platform independence BLESS can be deployed on any current computer architecture. Ease of usage ensures that the system can be used by non-experts.

Another special property of BLESS is the avoidance of single points of failures which could disrupt the functioning of the whole system. At the same time no omniscient entity is introduced which could be aware of all users or control their actions. Instead services are provided per building only. Also building data is only offered for the current building. This data can be used by MultiApp for localization, navigation, and for showing positions and routes on multi-level floor plans. All data and services thus stay in-house; there is no global Internet server collecting technical and internal details about a large number of buildings.

The implementation created as part of this dissertation closely follows the design. It allows buildings to offer services in a local and trusted way. It avoids massive, centralized collecting of user data, instead it allows the user to track what and when personal data is shared with the server part of the service.

8.2.2 Lessons Learned

In a field test the concept and implementation were examined. It was determined that the created system is well suited for providing private and

secure services that are bound to a specific location. Even though the field test was conducted in a private home, it is to be expected that BLESS could also be applied to larger buildings. However, three subjects need to be reconsidered.

Localization System

The applied localization system for positions at room-level caused serious issues. It often returned wrong rooms, resulting in geofence events to be reported falsely or late. It also caused a high battery drain. The presumed reason for these problems lies in the WiFi technology. WiFi was created as communication medium with high throughput. By no means it was intended to be used as accurate, energy-efficient localization technology. For enhancing the value of BLESS, future work should focus on integrating a better infrastructure-free localization technology which works reliably and allows power-saving, continuous indoor, room-grained tracking of users on client-side. Alternatively, even though associated with costs and potentially regular battery changing using Bluetooth beacons may be a valuable alternative. Costs could be kept low by not equipping the whole building but rather only for localization relevant areas.

Omniscient Entities

BLESS was compared to smart home system openHAB. Differences were outlined and vantages of BLESS opposed to smart home systems highlighted. During developing the test services for both BLESS and openHAB it became evident, however, that collecting and sharing data on server side – as done by openHAB – is helpful for smart home applications. In particular being aware of the presence status of users would facilitate the door guard services considerably. What is more, without the SP knowing if users are present, implementing the shutter down functionality from the openHAB version of the guard service is not possible with BLESS. The avoidance of omniscient entities for private homes thus seems unnecessary. Owners *must* trust their own system. Hence, systems must be developed, set up, and installed by trustworthy persons – either owners themselves or

a trusted service provider. Finally, users are mainly family members, which usually trust each other. Keeping e.g. location data hidden between users is not of high priority.

On the other hand, even if the design of BLESS avoids all central points of intelligence, consider shopping mall: Each shop offers its own service. Each shop thus knows when a user is present. There is no way for users to find out whether the SP of some shop services are linked. It is thus still possible to create a shopping and movement profile for each customer. The situation gets even worse by using (globally unique) user certificates. These allow to easily compare and match users' identities even across different shopping malls. As consequence it is recommended for future versions of MultiApp to create a new certificate for each building.

Background Actions

The field test also revealed a shortcoming in the implementation of MultiApp: Services are executed as JavaScript applications inside a WebView element. However, these elements are not intended to be used in background. As a result it is not possible to launch BLESS services completely invisible. Instead even when starting a service for a short background action, e.g. when invoking the door guard service to update the state of the door, the user will notice that the service opens and closes immediately afterwards. To eliminate this annoyance it is necessary to analyze the inner mechanisms of the WebView element. If not possible to execute JavaScript code without making the WebView visible, it may be necessary to include an additional JavaScript interpreter in MultiApp which allows executing JavaScript without GUI. This issue is considered to require the most effort to fix and should have a high priority when continuing working on MultiApp. Without fixing and with running many BLESS services, regular interrupts of the user's workflow are inevitable.

8.2.3 Outlook

Next to fixing mentioned open issues, the future of LLSs in general and of BLESS in specific strongly depends on the users. If people continue not

minding with whom they share their private data and instead trust every service provider blindly, LLSs will have no prospects. The drawback of LLSs is that they cannot be used out of the box. The price for staying in control and for keeping personal data in-house, is setting up the wanted LLSs first. This is laborious and costly. On the other hand, products of the big, global players, like Google and Apple, can be used without lengthy setup and often even without direct costs.

However, if the continuous reports about data-theft and espionage of private data from centralized Internet servers do affect users' actions, the demand for more decentralized services and products will grow. As a result shopping malls, airports, and cruisers could offer dedicated wireless networks for offering LLSs for their customers via BLESS. Skilled users could develop and set up their own LLSs and invite friends, e.g. for sharing their training progress in a friendly sports competition. It is even conceivable that such services are made public, e.g. via BLESS markets. Similarly to current app stores monetization and malware filtering features are possible. Even BLESS server containers are conceivable which help end users to easily download and set up services from such markets.

Bibliography

[Ais15] AISLELABS: *The Hitchhikers Guide to iBeacon Hardware: A Comprehensive Report by Aislelabs (2015)*. http://www.aislelabs.com/reports/beacon-guide/. Version: April 2015. – Accessed: 2016-08-08

[Apa13] APACHE: *Apache River*. http://river.apache.org/. Version: 2013. – Accessed: 2015-06-10

[App15a] APPLE: *App Store Review Guidelines*. https://developer.apple.com/app-store/review/guidelines/. Version: 2015. – Accessed: 2015-10-27

[App15b] APPLE: *App Store Review Guidelines – Privacy*. https://developer.apple.com/app-store/review/guidelines/#privacy. Version: 2015. – Accessed: 2015-11-09

[App15c] APPLE: *Local and Push Notification Programming Guide: Apple Push Notification Service.* http://developer.apple.com/library/mac/documentation/NetworkingInternet/Conceptual/RemoteNotificationsPG/ApplePushService/ApplePushService.html. Version: March 2015. – Accessed: 2015-06-01

[Apv14] APVRILLE, Axelle: *iOS Malware Does Exist*. https://blog.fortinet.com/post/ios-malware-does-exist. Version: June 2014. – Accessed: 2016-08-29

[Bar12] BARETH, Ulrich: Privacy-aware and energy-efficient geofencing through reverse cellular positioning. In: *Wireless Communica-*

tions and Mobile Computing Conference (IWCMC), 2012 8th International IEEE, 2012, S. 153–158

[BF+14] BOURQUE, Pierre ; FAIRLEY, Richard E. u. a.: *Guide to the Software Engineering Body of Knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014 http://www4.ncsu.edu/~tjmenzie/cs510/pdf/SWEBOKv3.pdf

[Bla14] BLACKBERRY LIMITED: *Push Service - BlackBerry Developer*. https://developer.blackberry.com/services/push/?CPID=PUSHAPI00. Version: 2014. – Accessed: 2015-06-01

[BN84] BIRRELL, Andrew D. ; NELSON, Bruce J.: Implementing remote procedure calls. In: *ACM Transactions on Computer Systems (TOCS)* 2 (1984), Nr. 1, S. 39–59

[Bre01] BREIMAN, Leo: Random forests. In: *Machine learning* 45 (2001), Nr. 1, S. 5–32

[BSNP+95] BAKHTIARI, Shahram ; SAFAVI-NAINI, Reihaneh ; PIEPRZYK, Josef u. a.: Cryptographic hash functions: A survey. In: *Centre for Computer Security Research, Department of Computer Science, University of Wollongong, Australie* (1995)

[CBR03] CHESWICK, William R. ; BELLOVIN, Steven M. ; RUBIN, Aviel D.: *Firewalls and Internet Security: Repelling the Wily Hacker*. 2. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2003. – ISBN 020163466X

[CDK05] COULOURIS, George F. ; DOLLIMORE, Jean ; KINDBERG, Tim: *Distributed systems: concepts and design*. Fifth edition. Pearson Education, 2005

[Chi11] CHITU, Alex: *Android Honeycomb's Browser Supports SVG*. http://googlesystem.blogspot.de/2011/02/android-honeycombs-browser-supports-svg.html. Version: February 2011. – Accessed: 2016-08-26

[CK13a] CHESHIRE, Stuart ; KROCHMAL, Marc: *DNS-Based Service Discovery*. RFC 6763 (Proposed Standard). http://www.ietf.org/rfc/rfc6763.txt. Version: Februar 2013 (Request for Comments)

[CK13b] CHESHIRE, Stuart ; KROCHMAL, Marc: *Multicast DNS*. RFC 6762 (Proposed Standard). http://www.ietf.org/rfc/rfc6762.txt. Version: Februar 2013 (Request for Comments)

[CL11] CHARLAND, Andre ; LEROUX, Brian: Mobile Application Development: Web vs. Native. In: *Queue* 9 (2011), April, 20:20–20:28. http://dx.doi.org/http://doi.acm.org/10.1145/1966989.1968203. – DOI http://doi.acm.org/10.1145/1966989.1968203. – ISSN 1542–7730

[DB07] DOWLING, Steve ; BARNEY, Amy: *iPhone Premieres This Friday Night at Apple Retail Stores*. http://www.apple.com/pr/library/2007/06/28iPhone-Premieres-This-Friday-Night-at-Apple-Retail-Stores.html. Version: June 2007. – Accessed: 2016-08-29

[Dee89] DEERING, Steve: *Host extensions for IP multicasting*. RFC 1112 (Internet Standard). http://www.ietf.org/rfc/rfc1112.txt. Version: August 1989 (Request for Comments)

[DH76] DIFFIE, Whitfield ; HELLMAN, Martin E.: New directions in cryptography. In: *Information Theory, IEEE Transactions on* 22 (1976), Nr. 6, S. 644–654

[eMa14] EMARKETER: *Worldwide Smartphone Usage to Grow 25% in 2014*. http://www.emarketer.com/Article/Worldwide-Smartphone-Usage-Grow-25-2014/1010920. Version: June 2014. – Accessed: 2015-11-09

[eMa16] EMARKETER: *Germany's Smartphone Market Surges Ahead*. http://www.emarketer.com/Article/Germanys-Smartphone-

Market-Surges-Ahead/1013634. Version: February 2016. – Accessed: 2016-08-29

[F-S14] F-SECURE: *THREAT REPORT H2 2013*. https://www.f-secure.com/documents/996508/1030743/Threat_Report_H2_2013.pdf. Version: 2014

[FCH+11] FELT, Adrienne P. ; CHIN, Erika ; HANNA, Steve ; SONG, Dawn ; WAGNER, David: Android Permissions Demystified. In: *Proceedings of the 18th ACM Conference on Computer and Communications Security*. New York, NY, USA : ACM, 2011 (CCS '11). – ISBN 978–1–4503–0948–6, 627–638

[FGW11] FELT, Adrienne P. ; GREENWOOD, Kate ; WAGNER, David: The effectiveness of application permissions. In: *Proceedings of the 2nd USENIX conference on Web application development*, 2011, 75–86

[FHE+12] FELT, Adrienne P. ; HA, Elizabeth ; EGELMAN, Serge ; HANEY, Ariel ; CHIN, Erika ; WAGNER, David: Android permissions: User attention, comprehension, and behavior. In: *Proceedings of the Eighth Symposium on Usable Privacy and Security* ACM, 2012, S. 3

[FHW85] FISCHETTI, Mark ; HORGAN, Jonathan ; WALLICH, Paul: The superstructure: Designing for high-tech: The high-tech house is hardly passive; its heating, security, communications, and lighting systems function independently, yet talk with one another at the command of computers. In: *Spectrum, IEEE* 22 (1985), Nr. 5, S. 36–40

[Ger11] GERMAN, Kent: *A brief history of Android phones*. http://www.cnet.com/news/a-brief-history-of-android-phones/. Version: August 2011. – Accessed: 2016-08-29

[GMR14] GANDHI, Anshuk ; MAGAR, Carmen ; ROBERTS, Roger: *How technology can drive the next wave of mass customization*. http://www.mckinsey.com/insights/business_

technology/how_technology_can_drive_the_next_wave_of_mass_customization. Version: February 2014. – Accessed: 2015-10-27

[Goo15a] GOOGLE: *Google Cloud Messaging for Android - Android Developers*. http://developer.android.com/google/gcm/index.html. Version: May 2015. – Accessed: 2015-06-01

[Goo15b] GOOGLE: *Google Play Developer Program Policies*. https://play.google.com/intl/ALL_us/about/developer-content-policy.html. Version: 2015. – Accessed: 2015-10-27

[Goo15c] GOOGLE: *Google Play Developer Program Policies – Personal and Confidential Information*. https://play.google.com/about/developer-content-policy.html#personal-confidential. Version: 2015. – Accessed: 2015-11-09

[Goo15d] GOOGLE: *Google Play Store – Facebook*. https://play.google.com/store/apps/details?id=com.facebook.katana. Version: November 2015. – Accessed: 2015-11-09

[Goo15e] GOOGLE: *Google Play Store – Google Play services*. https://play.google.com/store/apps/details?id=com.google.android.gms. Version: November 2015. – Accessed: 2015-11-09

[Goo15f] GOOGLE: *Google Play Store – WhatsApp*. https://play.google.com/store/apps/details?id=com.whatsapp. Version: November 2015. – Accessed: 2015-11-09

[Goo16] GOOGLE: *The Google Maps Geolocation API*. https://developers.google.com/maps/documentation/geolocation/intro. Version: August 2016

[GPVD99] GUTTMAN, E. ; PERKINS, C. ; VEIZADES, J. ; DAY, M.: *Service Location Protocol, Version 2*. http://www.ietf.org/rfc/

rfc2608.txt. Version: Juni 1999 (Request for Comments). – Updated by RFC 3224

[Heg12] HEGARTY, Christopher J.: GNSS signals - An overview. In: *Frequency Control Symposium (FCS), 2012 IEEE International* IEEE, 2012, S. 1–7

[Hir97] HIRSCH, Frederick J.: Introducing SSL and certificates using SSLeay. In: *World Wide Web Journal* 2 (1997), Nr. 3, S. 141–173

[Hof13] HOFFMAN, CHRIS: *iOS Has App Permissions, Too: And They're Arguably Better Than Android's*. http://www.howtogeek.com/?p=177711. Version: December 2013. – Accessed: 2016-08-29

[HT09] HANSEN, René ; THOMSEN, Bent: Efficient and accurate wlan positioning with weighted graphs. In: *Mobile Lightweight Wireless Systems*. Springer, 2009, S. 372–386

[HTTA13] HANSEN, René ; THOMSEN, Bent ; THOMSEN, Lone L. ; ADAMSEN, Filip S.: SmartCampusAAU – An Open Platform Enabling Indoor Positioning and Navigation. In: *2013 IEEE 14th International Conference on Mobile Data Management* Bd. 2 IEEE, 2013, S. 33–38

[HWJT10] HANSEN, Rene ; WIND, Rico ; JENSEN, Christian S. ; THOMSEN, Bent: Algorithmic strategies for adapting to environmental changes in 802.11 location fingerprinting. In: *Indoor Positioning and Indoor Navigation (IPIN), 2010 International Conference on* IEEE, 2010, S. 1–10

[Inf14] INFOWATCH: *Global Data Leakage Report 2014 – Number of registered data leaks, 2006-2014*. http://infowatch.com/report2014. Version: 2014. – Accessed: 2016-08-29

[Ins13] INSTEON ; INSTEON (Hrsg.): *Insteon Whitepaper: The Details*. http://cache.insteon.com/pdf/insteondetails.pdf. Version: October 2013. – Accessed: 2015-08-25

[JMV+12] JEON, Jinseong ; MICINSKI, Kristopher K. ; VAUGHAN, Jeffrey A. ; FOGEL, Ari ; REDDY, Nikhilesh ; FOSTER, Jeffrey S. ; MILLSTEIN, Todd: Dr. Android and Mr. Hide: fine-grained permissions in android applications. In: *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices* ACM, 2012, 3–14

[KBY+12] KIM, Ji E. ; BOULOS, George ; YACKOVICH, John ; BARTH, Tassilo ; BECKEL, Christian ; MOSSE, Daniel: Seamless integration of heterogeneous devices and access control in smart homes. In: *Intelligent Environments (IE), 2012 8th International Conference on* IEEE, 2012, S. 206–213

[Kjæ11] KJÆRGAARD, Mikkel B.: Indoor location fingerprinting with heterogeneous clients. In: *Pervasive and Mobile Computing* 7 (2011), Nr. 1, S. 31–43

[KM08] KJAERGAARD, Mikkel B. ; MUNK, Carsten V.: Hyperbolic location fingerprinting: A calibration-free solution for handling differences in signal strength (concise contribution). In: *Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on* IEEE, 2008, S. 110–116

[Koh78] KOHNFELDER, Loren M.: *Towards a practical public-key cryptosystem*, Massachusetts Institute of Technology, Diss., 1978

[KS07] KALOFONOS, Dimitris N. ; SHAKHSHIR, Saad: IntuiSec: a framework for intuitive user interaction with smart home security using mobile devices. In: *2007 IEEE 18th International Symposium on Personal, Indoor and Mobile Radio Communications* IEEE, 2007, S. 1–5

[Küp05] KÜPPER, Axel: *Location-based services: fundamentals and operation*. John Wiley & Sons, 2005

[KVV05] KRUEGEL, Christopher ; VALEUR, Fredrik ; VIGNA, Giovanni: Computer security and intrusion detection. In: *Intrusion De-*

tection and Correlation: Challenges and Solutions (2005), S. 9–28

[Lam71] LAMPSON, Butler W.: Protection. In: *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems* (1971), S. 437–443. – reprinted in ACM SIGOPS Operating Systems Review, 8,1, Janaury, 1974, pp. 18–24

[LBR+02] LADD, Andrew M. ; BEKRIS, Kostas E. ; RUDYS, Algis ; MARCEAU, Guillaume ; KAVRAKI, Lydia E. ; WALLACH, Dan S.: Robotics-based Location Sensing Using Wireless Ethernet. In: *Proceedings of the 8th Annual International Conference on Mobile Computing and Networking*. New York, NY, USA : ACM, 2002 (MobiCom '02). – ISBN 1–58113–486–X, 227–238

[LZYP13] LAOUDIAS, Christos ; ZEINALIPOUR-YAZTI, Demetrios ; PANAYIOTOU, Christos G.: Crowdsourced indoor localization for diverse devices through radiomap fusion. In: *Indoor Positioning and Indoor Navigation (IPIN), 2013 International Conference on* IEEE, 2013, S. 1–7

[Mau12] MAUTZ, Rainer: *Indoor positioning technologies*, Institute of Geodesy and Photogrammetry, Department of Civil, Environmental and Geomatic Engineering, ETH Zurich, Diss., February 2012. `http://e-collection.library.ethz.ch/eserv/eth:5659/eth-5659-01.pdf`. – Habilitationsschrift

[ME06] MISRA, Pratap ; ENGE, Per: *Global Positioning System: Signals, Measurements and Performance Second Edition*. Lincoln, MA: Ganga-Jamuna Press, 2006

[MG02] MUNSON, Jonathan P. ; GUPTA, Vineet K.: Location-based notification as a general-purpose service. In: *Proceedings of the 2nd international workshop on Mobile commerce* ACM, 2002, S. 40–44

[Mic15] MICROSOFT CORPORATION: *Push notifications for Windows Phone 8*. https://msdn.microsoft.com/library/windows/apps/ff402558(v=vs.105).aspx. Version: 2015. – Accessed: 2015-06-01

[Mil15] MILLER, Jack: *TCP connection between Android client and Windows server breaks after random time*. http://stackoverflow.com/questions/28212680/. Version: 2015. – Accessed: 2015-06-23

[MVH88] MAGAT, Wesley A. ; VISCUSI, W K. ; HUBER, Joel: Consumer processing of hazard warning information. In: *Journal of Risk and Uncertainty* 1 (1988), Nr. 2, S. 201–232

[Nel81] NELSON, Bruce J.: *Remote Procedure Calls*. Xerox, Palo Alto Research Center, Carnegie-Mellon University, USA, Diss., 1981

[PSH+09] PUIKKONEN, Arto ; SARJANOJA, Ari-Heikki ; HAVERI, Merja ; HUHTALA, Jussi ; HÄKKILÄ, Jonna: Towards designing better maps for indoor navigation: experiences from a case study. In: *Proceedings of the 8th International Conference on Mobile and Ubiquitous Multimedia*. New York, NY, USA : ACM, 2009 (MUM '09). – ISBN 978–1–60558–846–9, 16:1–16:4

[PTM14] PUDER, Arno ; TILLMANN, Nikolai ; MOSKAL, Michał: Exposing native device APIs to web apps. In: *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems* ACM, 2014, S. 18–26

[PwC14] PWC NETWORK: *Managing cyber risks in an interconnected world*. http://www.pwc.com/gx/en/consulting-services/information-security-survey/assets/the-global-state-of-information-security-survey-2015.pdf. Version: September 2014. – Accessed: 2016-08-29

[Ric14] RICHTER, Felix: *The Price Gap Between iOS and Android Is Widening*. http://www.statista.com/chart/1903/average-

selling-price-of-android-and-ios-smartphones/. Version: June 2014. – Accessed: 2015-11-09

[RM14] RIVERA, Janessa ; MEULEN, Rob van d.: *Gac Annual Smartphone Sales Surpassed Sales of Feature Phones for the First Time in 2013*. http://www.gartner.com/newsroom/id/2665715. Version: February 2014. – Accessed: 2016-08-29

[Rye99] RYE, Dave ; HOMETOYS.COM (Hrsg.): *Dave Rye @ X10*. http://www.hometoys.com/content.php?url=/htinews/oct99/articles/rye/rye.htm. Version: October 1999. – Accessed: 2015-08-25

[SC05] STEINBERG, Daniel H. ; CHESHIRE, Stuart: *Zero Configuration Networking: The Definitive Guide: The Definitive Guide*. " O'Reilly Media, Inc.", 2005

[Sch07] SCHMIDT, Stefan: *Service Location - Survey of mechanism to search and configure services*. http://datenfreihafen.org/~stefan/papers/university/service-location-paper.pdf. Version: 2007. – Accessed: 2015-06-10

[Sen15] SENIONLAB ; SENIONLAB (Hrsg.): *Indoor positioning 101*. https://senion.com/wp-content/uploads/2015/07/Technical-White-paper.pdf. Version: July 2015. – Accessed: 2016-08-05

[Sho05] SHORTY, Peter: *System and a method for building routing tables and for routing signals in an automation system*. April 12 2005. – US Patent 6,879,806

[SLG+12] SARMA, Bhaskar P. ; LI, Ninghui ; GATES, Chris ; POTHARAJU, Rahul ; NITA-ROTARU, Cristina ; MOLLOY, Ian: Android permissions: a perspective combining risks and benefits. In: *Proceedings of the 17th ACM symposium on Access Control Models and Technologies* ACM, 2012, 13–22

[SM94] STEWART, David W. ; MARTIN, Ingrid M.: Intended and unintended consequences of warning messages: A review and

synthesis of empirical research. In: *Journal of Public Policy & Marketing* (1994), S. 1–19

[SP13] SPAULDING, Elizabeth ; PERRY, Christopher: *Having It Their Way: The Big Opportunity In Personalized Products*. http://www.forbes.com/sites/baininsights/2013/11/05/having-it-their-way-the-big-opportunity-in-personalized-products/. Version: November 2013. – Accessed: 2014-04-27

[Sta15] STATISTA: *Number of apps available in leading app stores as of July 2015*. http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Version: July 2015. – Accessed: 2015-10-26

[SV04] SCHILLER, Jochen ; VOISARD, Agnès: *Location-based services*. Elsevier, 2004 https://books.google.de/books?hl=en&lr=&id=wj19b5wVfXAC&oi=fnd&pg=PP2&dq=location-based+services&ots=lcPp8zwiOp&sig=-HFa1bmFQw4yGAG9BZLqNOJRiO4#v=onepage&q=benefit&f=false

[TSFC10] TORRES-SOLIS, Jorge ; FALK, Tiago H. ; CHAU, Tom: *A review of indoor localization technologies: towards navigational assistance for topographical disorientation*. INTECH Open Access Publisher, 2010 http://cdn.intechweb.org/pdfs/9895.pdf

[UPn15] UPNP FORUM: *UPnP Forum*. http://upnp.org/. Version: 2015. – Accessed: 2015-06-10

[VF02] VALTCHEV, Dimitar ; FRANKOV, Ivailo: Service gateway architecture for a smart home. In: *Communications Magazine, IEEE* 40 (2002), Nr. 4, S. 126–132

[VTRB97] VIXIE, Paul ; THOMSON, Susan ; REKHTER, Yakov ; BOUND, Jim: *Dynamic Updates in the Domain Name System (DNS UPDATE)*. Internet RFC 2136, April 1997

[Was10] WASSERMAN, Anthony I.: Software engineering issues for mobile application development. In: *Proceedings of the FSE/SDP workshop on Future of software engineering research*. New York, USA : ACM, November 2010 (FoSER '10). – ISBN 978–1–4503–0427–6, 397–400

[Wil00] WILLIAMSON, Beau: *Developing IP multicast networks*. Bd. 1. Cisco Press, 2000

[Xia15] XIAO, Claud: *Novel Malware XcodeGhost Modifies Xcode, Infects Apple iOS Apps and Hits App Store*. http://researchcenter.paloaltonetworks.com/?p=10322. Version: September 2015. – Accessed: 2015-10-27

[XWZ14] XUE, Hui ; WEI, Tao ; ZHANG, Yulong: *Masque Attack: All Your iOS Apps Belong to Us*. https://www.fireeye.com/blog/threat-research/2014/11/masque-attack-all-your-ios-apps-belong-to-us.html. Version: November 2014. – Accessed: 2016-08-29

[ZGH07] ZHONG, Ge ; GOLDBERG, Ian ; HENGARTNER, Urs: Louis, lester and pierre: Three protocols for location privacy. In: *Privacy Enhancing Technologies* Springer, 2007, S. 62–76

[Zil15] ZILT, Hannes: *Service Discovery and JavaScript Remote Calls for Building-Linked Services using Smartphones*, Hamburg University of Technology, Institute of Telematics, Master's thesis, 2015

Author's Publications

[1] Julian Ohrt and Volker Turau. Cross-platform development tools for smartphone applications. *IEEE Computer*, 9(45):72–79, September 2012.

[2] Julian Ohrt and Volker Turau. Building-linked location-based instantaneous services system. In *Proceedings of the 5th International Conference on Ambient Systems, Networks and Technologies (ANT2014)*, June 2014.

[3] Julian Ohrt and Volker Turau. Simple indoor routing on svg maps. In *Indoor Positioning and Indoor Navigation (IPIN), 2013 International Conference on*, pages 1–6. IEEE, October 2013.