POLITECNICO DI MILANO
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

SPF: Social Proximity Framework

A Middleware for Social Interactions
in Mobile Proximity Environments

Relatore: Prof. Sam Jesus GUINEA MONTALVO
Correlatore: Ing. Valerio PANZICA LA MANNA

Tesi di Laurea di:
Jacopo ALIPRANDI, matricola 799010
Dario ARCHETTI, matricola 798847

Anno Accademico 2013 - 2014

Ai nostri genitori

Abstract

The advent of Web 2.0 and social networking applications was a major technological shift that changed the web from an internet of documents to an internet of people, with a novel focus on user identity. Nowadays we are facing another technological shift that regards the world of the Internet of Things. Smart interconnected devices are changing our lives, and turning conventional spaces into smart ones, where interactions are supported by digital means. This change is fostered by the development of new networking technologies that aim at supporting device-to-device communication. The industry efforts, however, neglect the social identity of users, which was a key enabler of the Web 2.0. As result, smart solutions that want to integrate the social dimensions need to rely on external infrastructures and integrate different social networking services.

We believe that, in the context of smart spaces, the social dimension and the physical proximity can be exploited to create new social proximity services. This thesis presents the Social Proximity Framework (SPF), a software solution for the creation of social smart spaces, where social identity and proximity is taken into account to offer personalized services. The framework offers benefits both to users and application developers. Users can enjoy richer services and personalized content, along with means to control their personal information contained in the framework. Developers can leverage the tools offered by the framework to ease the development of their applications, and exploit the data of user profiles generated by different social applications. SPF can also be adapted to multiple networking technologies thanks to its abstract communication layer. Two technologies have been used to study how SPF can benefit from emerging device-to-device communication protocols.

Sommario

L'avvento del Web 2.0 e delle applicazioni di social networking è stato un notevole progresso tecnologico che ha trasformato il web da una rete di documenti in una di persone, mettendo al centro l'utente e la sua identità. Al giorno d'oggi stiamo assistendo ad un altro progresso tecnologico che riguarda il mondo dell'Internet of Things. L'introduzione di dispositivi intelligenti interconnessi tra loro sta cambiando le nostre vite, trasformando spazi convenzionali in spazi intelligenti, dove le interazioni sono supportate da mezzi digitali. Questo cambiamento è favorito dallo sviluppo di nuove tecnologie di rete, che puntano a supportare la comunicazione diretta tra dispositivi. Le soluzioni industriali, tuttavia, trascurano l'identità sociale degli utenti, un fattore fondamentale per il Web 2.0. Di conseguenza, applicazioni *smart* che vogliono integrarsi con la dimensione sociale dell'utente devono affidarsi su infrastrutture esterne e integrarsi con molteplici servizi di social networking.

Crediamo che, in un contesto di *smart spaces*, la dimensione sociale e la prossimità fisica possono essere sfruttate per dare vita a nuovi servizi di *social proximity.* Questa tesi presenta *Social Proximity Framework* (SPF), un prodotto software per la creazione di *social smart spaces* in cui l'identità dell'utente e la vicinanza fisica vengono sfruttate per offrire servizi personalizzati agli utenti. Il framework offre benefici sia agli utenti sia agli sviluppatori: gli utenti possono fruire di servizi più ricchi e di contenuti personalizzati, insieme a modalità di gestione dei dati personali inseriti nel framework. Gli sviluppatori possono invece sfruttare gli strumenti offerti dal framework per semplificare lo sviluppo delle loro applicazioni e accedere all'informazione contenuta nei profili degli utenti, proveniente da molteplici applicazioni. SPF può essere adattato a diverse tecnologie di rete grazie al suo middleware di comunicazione astratto. In particolare, sono state utilizzate due tecnologie di rete diverse per studiare come SPF possa trarre beneficio da tecnologie di comunicazione *device-to-device* emergenti.

Ringraziamenti

Alla fine di questo lungo percorso ci sono molte persone che vogliamo ringraziare: per il supporto durante lo svolgimento di questo lavoro di tesi e più in generale per averci accompagnato in questi anni. Inevitabilmente dimenticheremo qualcuno. Grazie innanzitutto a Sam Guinea, relatore di questa tesi, per l'immancabile supporto e fiducia che ha riposto in noi. Grazie a Valerio Panzica La Manna per averci coinvolto e aiutato in questo interessante progetto. Grazie al prof. Luciano Baresi per la costante partecipazione e stima che ci ha dimostrato. Grazie a Telecom Italia e allo SCube Joint Open Lab per averci accolto nel laboratorio e fornito interessanti spunti di riflessione. Grazie agli amici, universitari e non, per averci sopportato (non è un errore di battitura) durante questo periodo di studio al Politecnico. Grazie infine ai nostri genitori, a cui è dedicata questa tesi, per averci accompagnato in questi 25 anni.

Jacopo e Dario

Non posso non ringraziare i miei compagni di avventure Isaia, Andrea, Roberto per il lavori svolti assieme e le lunghe chiaccherate. Grazie a Chiara Cerletti per aver cercato i puntini sulle i di questo lavoro. E infine, ma non per valore, Serena, Alberta, Martina e Micol perché nonostante non ci si veda mai è sempre come essere a casa. Manca solo Dario: grazie per avermi pazientemente sopportato!

Jacopo

Ringrazio Gaia, perché mi è vicina sempre, e mi sostiene e motiva come solo lei sa fare. Ringrazio Chiara, compagna di progetti e di studio, per avermi spinto a studiare anche quando la voglia mancava. Ringrazio Jacopo, per avermi accompagnato in questo lavoro di tesi, impegnativo ma gratificante.

Dario

Contents

Chapter 1

Introduction

In recent years the advent of Web 2.0 and social networking applications was one of the major technological shift that profoundly changed our lives. This revolution was fostered by the spread of very popular applications that were able to attract millions of users, changing the web from an internet of documents to an internet of people. Social identity is commonly seen as a mean to enable communication between people and to provide services tailored to the user needs, preferences, and interests.

Nowadays we are facing another technological shift that regards the world of the Internet of Things. The adoption of smartphones and smart interconnected devices is changing the way in which we interact with physical spaces, which become day by day smarter. This change is fostered by the development of new networking technologies that aim at supporting innovative scenarios, as well as solving the issues that come from the particular nature of mobile devices. In mobile proximity environments, device-to-device communication is seen as a promising solution to overcome the need of external infrastructure. In this field, the industry efforts have produced examples of connectivity frameworks that aim to provide a higher level of abstraction over the network technologies: the focus of these solutions is on the management of the heterogeneity of transport protocols and operating systems.

Unfortunately, the social identity is often neglected and never taken into account to create innovative services. As result, smart solutions that want to integrate the social dimensions need to rely on external infrastructures and face problems that arise from the integration of different social networking services.

We think that even in smart spaces the social dimension and the physical proximity, enabled by device-to device communication, can be exploited to create new *social proximity services*. The aim of this thesis is to fill this gap by proposing the **Social Proximity Framework** (SPF) as a software solution for the creation of *social smart spaces*, where social identity and proximity is taken into account to offer personalized services and support real-life interactions with digital ones.

During the design of SPF we took as reference model the state of the art regarding web social networking applications. The existing social network sites available on the Internet are all based on centralized, isolated systems, and usually each of them is run by a single company. Users on one social network cannot interact with users on another social network. As result, people will often have to sign up for an account on multiple web sites to keep in touch with different groups of friends. Here the term *walled garden* is used to denote such services that use proprietary interfaces that prevent the social data from being shared across the network, creating a wall around them [8].

In this field, several industrial works [9, 10, 15] aim at providing a unified model for the integration of social networking services. The contents of these works can be classified according to two main objectives: the first is to provide a common interface for developers to build application on the top of social networking services [9, 10]; the second is to define an infrastructure to allow the federation between different social networks[15].

When we tried to move these concepts from the web to the mobile area, we realized that the personal nature of mobile devices and the more complex scenarios enabled by smart spaces would have required a different approach. While in the web users are seen as contents of a social networking service, in the smart space the user has to be brought into focus. This user-oriented approach allows the user to manage the services he uses, according to the context and to privacy settings.

To understand the potentiality of the novel scenarios that device to device communication offers, we analyzed some of the most recent networking technologies as well as innovative solutions that will be available in the near future.

Our main reference and first choice for the implementation of the network layer was AllJoyn [5]. AllJoyn is an open source middleware created by Qualcomm in 2011. It is a platform neutral solution that aims to provide a comprehensive framework for deploying proximity-based distributed applications on heterogeneous systems with mobile elements. The technology offers the developers a classic distributed object oriented system, which is enriched by features, like advertising and discovery, that are typical of mobile-proximity contexts.

Unfortunately, the current implementations of AllJoyn supports only standards Wi-Fi networks, so that there is the need of a fixed Access Point (AP). On the other hand, Wi-Fi Direct provides a solution to this issue by assigning dynamically the role of AP, therefore it is suitable for mobile and outdoor scenarios characterized by the absence of a network infrastructure. Another promising solution in the field of device-to-device communication is LTE Direct [7], which provides a scalable service discovery over the LTE licensed spectrum.

When these networking technologies meet the social dimension, there are two scenarios that show up as the most relevant. The first one regards the possibility of social networking without requiring external infrastructure. In this scenario users can discover new people in proximity and interact with each other. Some existing examples of such applications have started to emerge. Bizzabo[1], based on AllJoyn, is an event management platform that allows event organizers to maximize attendee engagement with one-on-one mobile and proximal messaging. FireChat[2] is a mobile app that allows its users to chat without the need of an internet connection.

The second scenario is the one related to targeted advertising, where owner of shops, bar or restaurant can deliver local and targeted advertisements on the basis of the interests and habits of their customer. Even here the idea of using device-to-device communication technologies to improve customer experience is not new. Bluetooth LE beacons are used as indoor positioning system and to add proximity awareness to applications, but the integration of social data requires an external

[1]`https://www.bizzabo.com`

[2]`https://opengarden.com/firechat`

infrastructure.

The idea of social smart space is obviously not limited to these two application scenarios, nevertheless they are sufficiently detailed with regard to the kind of interactions that should be supported. The analysis of these scenarios led us to outline a series of high-level functions that should be provided by the framework. These constitute the building blocks available to developers to realize their own *social proximity applications*.

The aim of SPF is to embrace the world of web social networks and the opportunities enabled by device-to-device communication. The SPF contributes to the creation of social smart spaces in different ways. It facilitates, speeds up, and reduces the complexity of the development of novel proximity-based services by abstracting reusable functionality. The SPF provides well-defined interfaces to let devices (people) advertise and share identities and exploit each other's services. The SPF facilitates the dynamic deployment of new services, and thus the continuous modification of existing spaces to take into account new participants and needs. User-oriented applications, at the same time, can exploit a clear, privacy-friendly solution that supports different profiles and permissions based on habits, contexts, or place of use.

The result is a software solution where both users and developers may find advantages. Users may benefit from richer services and personalized services along with means to control them. Developers can leverage on the offered tools to ease the development of their applications and exploit the data of a social profile generated by different social applications.

Conceived and implemented for Android device, SPF is a framework built on the top of an abstract communication middleware that provides an integrated view over the social dimension of the user. By relying on the Android inter app communication framework, SPF allows applications to define social proximity services, contribute to the creation of the user profile and access communication functions, reinvented with a social flavor.

A software library mediates the access to the framework by offering reusable components that are tied to the life-cycle of the operating system components, and thus they are easier to manage than other networking solutions. Moreover, the centralized solution allowed us to define a more optimized use of the resources and to take charge, without the intervention of the external application, of all those operations that are required to advertise the service over the network and to keep it active.

Two different versions of the communication middleware, based on AllJoyn and Wi-Fi Direct, help explain the characteristics of the different components, and show how the SPF can benefit from emerging device-to-device communication technologies.

The thesis is structured as follows:

- Chapter 2 discusses the state of the art of mobile device-to-device technologies as well as some industry works that aim at integrating web application and social networks. A short review of mobile social middleware from the academia helps to outline some of the problems that have been already studied in this area.

- Chapter 3 describes the functionality of SPF. It starts with a discussion about the scenarios and high level requirements, then it continues with the definition of the building blocks for social proximity applications and how these can be managed by the user.

- Chapter 4 describes the architecture of SPF. Here we discuss the interfaces of SPF and its internal components. The chapter ends with a description and a discussion about two implementation of the networking layer based on AllJoyn and Wi-Fi Direct.

- Chapter 5 discusses the realizations and the evaluation of SPF. It explains how the tools offered by the framework can be used to implement the scenarios discussed in Chapter 3. Then it presents our consideration about the improvement of SPF on the code quality of applications. Finally an assessment on the performance and limitations of the inter-process architecture concludes the chapter.

- Chapter 6 discusses the final results of the SPF and outlines some future works regarding emerging network technologies, the porting on other operating systems, and features that can extend the current implementation of SPF.

- Appendix A provides the guide to the API of SPF library that can be used by developers to implement SPF-enabled applications.

- Appendix B provides the guide to the internal API of the framework. It explains how SPF can be integrated in other solutions and how to implement a different version of the communication middleware.

A paper titled *"SPF: A Middleware for Social Interaction in Mobile Proximity Environments"*, related to this thesis, has been accepted as contribution to the *37th International Conference on Software Engineering* (ICSE 2015).

Chapter 2

State of the art

Interconnected devices are changing the way in which we interact with physical spaces, that are becoming day by day smarter. This change is fostered by the wide diffusion of Wi-Fi networks as well as several emerging mobile technologies that enable seamless interactions between devices. Unfortunately, when talking about smart spaces, the key enabler of the Web 2.0 is neglected: the social element. This is often seen as an accidental information and is not exploited to create new *social proximity services*.

This chapter describes the technologies and the theoretical background on which this thesis is based. The first section provides an overview of the networking technologies and middleware for implementing mobile proximity services. The second section discusses the social networking models as they appear in the web, along with the industry efforts to provide integration solutions across different services. The third section presents several different works, developed in academic contexts, that aim at enabling social interactions in mobile proximity environments. Finally, the fourth section introduces the problem addressed in this thesis.

2.1 Proximity networking technologies

Recent years have seen the proliferation of smart devices from simple wireless sensors to more advanced products like smartwatches or TVs. Given their widespread adoption, smartphones became the natural controllers of these smart objects.

When trying to connect these devices, the most standard approach is based on centralized server solutions that leverage the existence of local Wi-Fi networks. Despite this is a well-known and consolidated approach, it suffers from limitations that comes from the need of internet connectivity: it may not be applicable in outdoor scenarios where connectivity is not available, it has high latency due to remote communications and may not fit the energy requirements of battery supplied devices.

On the other hand, direct device to device communication presents problems and design issues that arise from the heterogeneity of devices and from the existence of several different technologies. These solutions differentiate themselves by targeting various contexts of adoption as well as requirements like energy efficiency, scalability, transmission range and speed. This situation does not become clearer even if we restrict the area to the interaction between two smartphones. Here there is the need to deal with different versions of operative systems, different hardware availability, and proprietary technologies that prevent the spread of proximity based mobile applications.

Differently from what happens in web development, where myriads of high level frameworks are available, mobile development lacks middlewares that may provide a higher level of abstraction for implementing services. Once selected the proper technology, a developer has to deal with the complexities of the networking solution, which usually implies to design all the software layers from the socket-like API provided by the operative system, to the business logic of the service.

This section presents an overview of the most recent networking and middleware technologies for implementing proximity services, as well as promising industry solutions that will be available in the near future.

2.1.1 Wi-Fi Direct

Wi-Fi Direct, also named Wi-Fi P2P, is a technology defined by the Wi-Fi Alliance[1] in 2010 that provides peer-to-peer communication over Wi-Fi. For long Wi-Fi networks were limited to the basic model of an Access Point (AP) creating the wireless network and devices connecting to it. Even if the IEEE 802.11 standard allows device to device connectivity by means of the *ad-hoc* mode of operation, this is affected by limitations in the requirements, e.g., lack of power saving support and extended QoS capabilities. Another extension to the standard is 802.11z, namely Tunnel Direct Link Setup (TDLS), which enables device to device connectivity but requires the devices to be connected to the same AP[2].

The major novelty of Wi-Fi Direct is that it works without the need of a fixed Access Point, since the roles are dynamically assigned. Moreover, it offers power saving support and provides higher speed and range than the ones achievable with communication over Bluetooth. As result, Wi-Fi Direct is suitable for battery powered devices and applications that need to share data at high rate, without requiring an internet connection.

In typical Wi-Fi networks, access points create and announce wireless networks, while clients scan and associate to them. To overcome the need of a fixed AP, in Wi-Fi Direct these roles are specified as dynamic, and hence devices have to negotiate who will take over the AP-like functionalities. The device that takes the role of the AP is called *P2P Group Owner*, and is in charge of providing a software access point (*soft AP*) that the other devices, named *P2P Clients*, can use to establish connections as in standard Wi-Fi networks. When a *P2P Group* is created, the devices form a star-shaped network topology where the assigned roles are fixed; therefore, in case of disconnection by the P2P Group Owner, its role can not be transferred to another P2P Device and the connections with all the other peers are closed.

The technology specifies different types of group formation techniques; in this paragraph we first discuss the *Standard* procedure, where P2P Devices have to discover each other and then negotiate which device should act as P2P Group Owner. In this case Wi-Fi Direct devices start by performing traditional Wi-Fi scan to discover existent P2P Groups; a P2P Group is announced by the P2P Group Owner through beacons like a traditional AP. Then a discovery algorithm is executed,

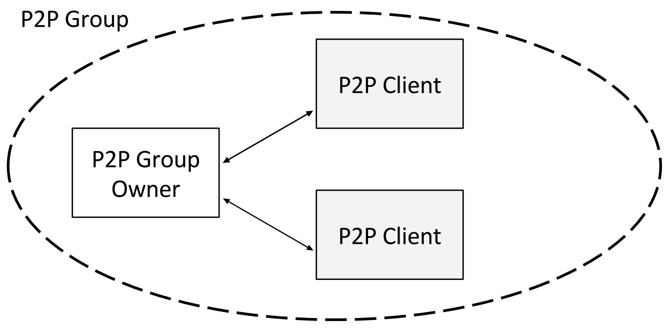


Figure 2.1: Wi-Fi Direct P2P Group topology

allowing two P2P Devices to find each other by sending *probe requests* and receiving *probe responses*. Once the two devices have found each other, they can start the negotiation phase which consists in a three-way handshake (GO Negotiation Request/Response/Confirmation). In order to reach an agreement, the devices send a numerical parameter, namely GO Intent value; the device that sends the highest value is elected as group owner. In case of same GO Intent the owner is selected randomly. At the end of this procedure they have agreed on which device will play the role of P2P Group Owner.

There are other two types of group formation procedure that are simplified version of the *Standard* technique, called *Autonomous* and *Persistent*. In the *Autonomous* group formation there is no negotiation phase: a device can decide to create a P2P Group autonomously and therefore becomes the group owner, by selecting a transmission channel and starting to beacon network information. In the *Persistent* case, the network credentials and the P2P roles are persistently stored on the devices, and thus they can be used to speed up subsequent re-instantiation of the same group.

An important feature of Wi-Fi Direct is the ability to support service discovery at the link layer. This allows two devices to decide whether to connect or not prior to the group creation process. The Service Discovery procedure is an optional frame exchange that can be performed at any time between any discovered P2P Devices. It is based on the Generic Advertisement Service (GAS) protocol defined in IEEE 802.11u[3], that supports higher-layer advertisement protocols employing a query/response mechanism such as Bonjour and UPnP.

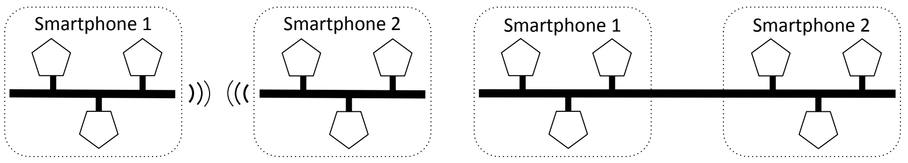
Android operating system introduced Wi-Fi Direct support starting from API Level 14[4]. The low level details of the technology is completely transparent to the application developer who can access the functionality through its dedicated API. This API supports the advertising and discovery of P2P Devices as well as of the services they offer. In the latter case, service advertising requires some additional information: the name of the service, the type in the form of *protocol._transportlayer*, and an additional key-value map of strings called *records*, which can contain additional information about the service. Unfortunately, the size of these records cannot exceeds 512 bytes, thus preventing the advertising of a relevant amount of information.

Once the devices or services are discovered, results are delivered to the application by means of callbacks following an asynchronous pattern. According to the received information, the application may decide to start the negotiation process by means of a *connect* operation. Before joining a group, the devices must be authenticated: usually this happens through WPS. This procedure requires the acceptance of a predefined system pop-up dialog, which is prompted to the user of the designated P2P Group Owner device. Once the connection is established the P2P Clients are notified with a callback about the P2P Group Owner IP address that can be used to communicate with standard sockets. Further changes about the Wi-Fi connection can be received with a BroadcastReceiver, listening to proper system Intents.

2.1.2 AllJoyn

AllJoyn[5] is an open source middleware created by Qualcomm in 2011. It is a platform neutral solution that aims to provide a comprehensive framework for deploying proximity-based distributed applications on heterogeneous systems with mobile elements. The technology offers developers a classic distributed object oriented system, which is enriched by features, like advertising and discovery, that are typical of mobile-proximity contexts. Moreover, it handles the complexities of dealing with different network technologies by providing dynamic configuration and abstracting out the details of the physical transports.

The basic abstraction of the AllJoyn system is the AllJoyn *Bus* or *Router*. This component is in charge of moving the messages around the distributed system: the goal is to allow two applications to communicate without dealing with the underlying mechanism. Communication can be local, when the two applications run on the same device, or distributed. In the latter case, AllJoyn takes care of unifying the two remote Bus segments according to the available physical transports. This is completely transparent to the users of the distributed bus, as it appears as if it is local to the device. Figure 2.2 depicts the AllJoyn Bus abstraction where pentagons represent the connections of the applications that are attached to the bus.



(a) Alljoyn connects the bus segments of different devices.

(b) The distributed bus appears as if it is local to the devices.

Figure 2.2: Alljoyn distributed bus

There exist three different topologies with regard to the interaction between the AllJoyn Router and applications:

- in the *Bundled Router* configuration, applications use their own AllJoyn Router which is bundled within the app. Generally this is the case of mobile OS like Android and iOS and desktop OS like Mac OS X and Windows.
- in the *Standalone Router* configuration, multiple apps connect to a same router which is hosted by an external background/service process. This is common case of Linux systems where the Router runs in an external daemon.
- finally, embedded device can leverage a thin version of the framework where the Router is hosted on a remote device.

Figure 2.3 shows an example of these three different topologies. On the left there is a smartphone with the bundled configuration, on the right there is a desktop with a standalone router, while the two embedded device are connected to their AllJoyn routers by means of a thin version of the framework library.

Applications can connect to the AllJoyn router by means of a Bus Attachment. This is a language specific component offered by the API of the framework that represents the distributed bus. A bus attachment can be used to register and advertise services as well as to discover remote ones. The bus assigns a temporary unique name to each

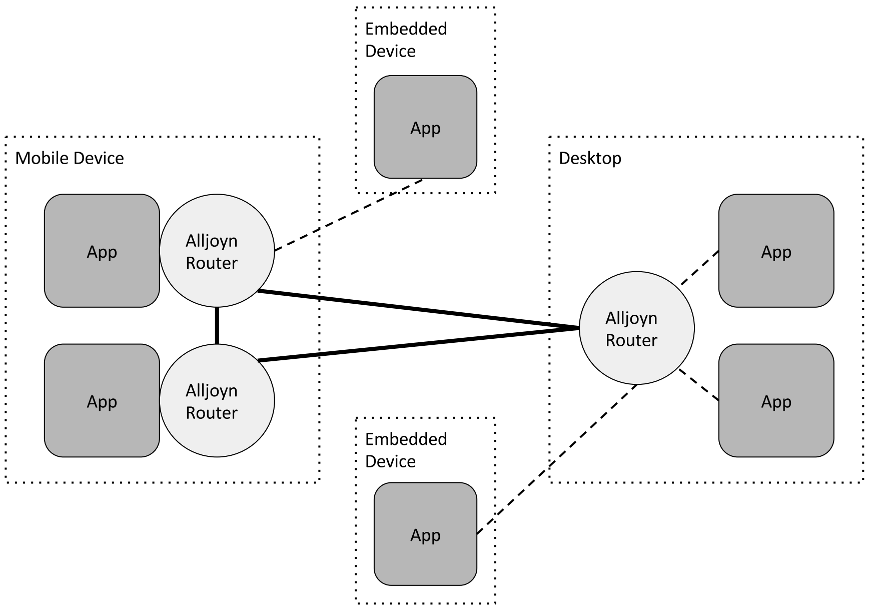


Figure 2.3: AllJoyn Router topologies

attachment; if the application needs to provide a service, it can ask for a persistent name, called *well-known name*. Well-known names are announced on the distributed bus and thus can by used by remote applications for service discovery.

As AllJoyn is a distributed object oriented system, it allows applications to register remote objects, that take the name of *bus objects*, whose methods can be invoked by remote clients. As classical in object-oriented programming, bus objects may have methods and properties that are known as *bus methods* and *bus properties*. Moreover, AllJoyn introduces the concept of *bus signal*, which is an asynchronous notification that can be broadcast to signal an event or a change in the state of an object. To be remotely invoked, all the members of a bus object must be declared in an interface which is used as contract to the external world and for the marshalling and unmarshalling of parameters.

Bus objects are registered and reside within a bus attachment, multiple implementations of a same service can be addressed with different object paths. Well-known name and object path are the keys used by the framework to deliver and route messages: the first is used within the AllJoyn Router while the latter identifies the communication endpoint within a bus attachment.

Clients can access the remote service through a remote method invocation mechanism. Once they have found a well-known name of the service they want to access, they can ask the bus attachment to open a session with the remote provider. At this point, the AllJoyn Router performs all the operations needed to establish a connection with the remote peer. Finally, to invoke the remote method, the application has to obtain from the bus attachment, a proxy object that implements the interface of the service. This object, called *proxy bus object* is in charge of marshalling the invocation request and delivering the message to the local connection to the bus.

2.1.3 Intel CCF

Intel Common Connectivity Framework[6] is a proprietary middleware developed by Intel, that aims at providing a cross-platform solution for building peer to peer applications without dealing with the heterogeneity of networking technologies. As its aim is the same one of AllJoyn, Intel CCF shares many features with the previously discussed middleware, but differentiate itself by introducing social identity in the discovery process and providing cloud services to support the creation of P2P networks.

Differently from what happens in AllJoyn where the service discovery is based on simple string identifiers (well-known names), Intel CCF requires the setup of user identities before a service can be made discoverable. This step consists in the specification of a user name, an avatar and a device name. Once the communication channel is established, the API provides a stream based communication, different from the RPC model proposed by AllJoyn. Moreover, CCF supports fail-over mechanisms that provide automatic re-connection even on different physical transports.

Intel CCF cloud functions deal mostly with networking issues such as NAT and firewall traversal. Among these functions, the most interesting for the development of social applications is the concept of *discovery nodes*. A discovery node is a cloud component that applications can use to exchange discovery data by publishing and subscribing to it. A node can represent different things such as a location, a group of friends, and can be used in different contexts to filter and enrich the standard discovery process.

At the moment of writing, CCF is in a beta phase and plans to support Wi-Fi and LAN networks, as well as Bluetooth, Wi-Fi Direct and cellular connections.

2.1.4 LTE Direct

LTE Direct[7] is an innovative device-to-device discovery technology that utilizes the LTE licensed spectrum and is part of the Release 12 of the 3GPP standard. Its aim is to scale up from existing proximal discovery technologies in a battery efficient and privacy sensitive manner.

Existing approaches to proximal discovery provide valuable solutions for very specific use cases but fail when trying to provide energy efficient, scalable and always-on proximal discovery services. Location-based approaches track the user location and use a centralized solution to determine the proximity to a given area, e.g., geofencing. This approach is excellent for the unlimited range and the large install base, but the battery consumption, caused by the periodic location tracking and the network accesses, limits its efficiency when trying to provide an always-on discovery service.

On the other hand, Bluetooth LE proximity beacons utilize device-to-device discovery to advertise services to nearby users through a pre-installed app. Proximity beacons are an energy efficient approach for providing location awareness and geo-fencing but their use is limited to specific scenarios. In fact, the beacons are able to broadcast in a range of approximately 50 meters; moreover, they do not scale well with regard to energy consumption. Wi-Fi Direct can offer a similar solution, but even in this case the battery life limits the scalability of the solution. This is primarily because these technologies operate in the unlicensed spectrum and have to deal with uncontrolled interferences coming from other devices.

LTE Direct is designed to overcome the limitations of the existing approaches in a battery efficient, scalable and interoperable way. Mobile applications can use the technology to monitor application services on other devices or broadcast their own services. LTE Direct enabled devices can broadcast these information via beacons, that are sent periodically leveraging the LTE network for timing and resource allocation. These beacons, called *Expressions* are 128 bit identifiers that can represent different things such as an identity, a service, an interest or a location. The relevancy of Expressions is determined at the device level and filtered at the physical layer. Therefore there is no need for an applications to be active, but it is notified when LTE Direct detects a match, according to a monitor previously set. Expressions can be targeted to specific applications or public, in the latter case a common language is provided to ensure interoperability. Mobile applications can map their services to public Expressions by means of a centralized Expression Name Server that contains a hierarchy of interest categories.

2.2 Web and social services integration

The introduction of the Web 2.0 technologies has radically transformed the World Wide Web into a social space, shifting the focus from documents to people: individuals were no more mere consumers of content, but rather active publishers of information that other peers can interact with. As discussed in [8], this technological shift, together with the human predisposition for interaction, has led to the blossoming of on-line social networking services, providing real-time communication capabilities. The most notable example is Facebook, a social networking service founded in 2004 that counts 1.39 billions users as of December 31st, 2014[1].

Most of the social networking services available nowadays share a common set of features that are familiar to their users. Here we provide a list of the most characteristic features of general purpose social networks.

Identity Entities and resources are assigned an identifier, unique within the system, thus enabling addressability. In this way they can be referenced from other entities, within the social network or from the outside, using a machine-processable address.

Profile Each user of a social network is assigned a profile, which is a container of data describing various personal details: name, profile picture, address, ... Users may restrict the access to their personal detail to a subset of users.

Relationships Users can create links of various nature with other users on the same social network; these relationships can be mutual, e.g., “friendship”, and typically require both party to confirm the tie, or one-sided, e.g., “follower”. The profile of a user

[1]http://newsroom.fb.com/company-info/ (cited January 2015)

often displays his relationships, and the access to this piece of information can be restricted, similarly to the other profile details.

Search Users can search for other people specifying one ore more personal details, including name, location, age or gender; some services allow users to opt-out of the search service to be excluded from search results. The search may also include other kind of resources, like events and photos, depending on the type of social network.

Content sharing Users share digital content on social networks to make it available to their list of contacts. The submitted content may be an original contribution, like a photo taken by the user, or a content originally created by someone else; in this case the original author is normally credited. Similarly to profile details, users can restrict access to their shared content so that only a subset of users can access it.

Activities One of the most characteristic feature of the social web is the real-time sharing of *activities*, i.e., social actions between users or between users and resources. Social networking services support multiple types of activities: some are generic and widely supported, like a status update or the upload of a photo, others are more context-specific. The collection of all the activities performed by a user is named the *stream* of that user. Generally, social networks support follow-up actions, performed by users as a response to an activity of another user: the most famous is the "like", used to express appreciation.

Groups Social networks may allow the creation of groups, i.e., subsets of users that may share a common interest. Users can join or leave existing groups, possibly upon the approval of the administrator, and share messages or content with the other members of the group. The administrator may provide a profile for the group, similar to that available for users, and create events or other resources contextually to the group.

Private messages Users can send private text messages that can be read only by the sender and by the receiver; some services allow the creation of group conversations, where only members can exchange messages. It may also be possible to send other types of resources, like photos or files.

While the web as a network of people is becoming more and more interconnected, most of the social networking services available on the internet are designed as centralized, isolated systems with minimal interoperability, if any. A new term, *walled garden*, has been coined to denote a service in which its provider has control over content and interactions, and restricts access to its convenience: normally, social networking services allow data to be inserted straightforwardly, while it can be accessed and manipulated only by means of proprietary interfaces and data formats, both for humans and machines.

For the end user, this means that the interaction with users of other services is difficult if not impossible, and that a new account is often required to start using a new service. This lack of common standards also affects application developers, who need to implement service-specific connectors, as each social network exposes a different interaction interface. These drawbacks has fostered multiple attempts for the creation of open standards to enable interoperability and federation among heterogeneous services.

In the following, we describe three examples of specification for social interactions: Activity Streams, which defines a standard for the description of user activities, OpenSocial, which provides a standard interface for the development of applications interacting with social services, and SNeW, which defines a standard architecture for a system of interoperable and federated social networks.

2.2.1 Activity Streams

Activity Streams [9] is a specification that introduces a standard format for describing social actions, performed by users in the context of a social networking service. The adoption of a common standard enables the interaction among different social networking services and provides a ready to use model that avoids the development of an ad-hoc solution.

The basic element of the specification is the *activity*, which describes an unique and identifiable, completed or potential social action, that is performed by an actor over an object, in the context of a social networking service. An example of activity is a status update on the Facebook wall, as well as a particular achievement in a game. In case of potential actions, the activity describes what the actor may do with the object, while in case of completed actions the activity describes what has been done. An *Activity Stream* is a collection of activities, which may have some details in common; the Facebook wall and the Twitter timeline of a user are examples of activity streams.

The aim of the specification is to provide a standard syntax, based on the JSON format, that is rich enough to describe activities in a human-friendly, yet machine-processable and extensible manner. Each activity is characterized by a *verb* and a series of properties that define the type of an activity and provide contextual information. In particular, in addition to an actor, a verb and an object, an activity may feature a target, which is the recipient of the activity, a result, which is a web resource created as a result of the activity, and participants, one or more people that participated in the activity.

2.2.2 OpenSocial

The proliferation of social websites forced application developers to choose which ones to write applications for. Moreover, each social website provides its unique APIs, which are tied to the application context of the platform and often does not follow any shared standard. As result, this limits the spread of new applications and increases the efforts by the developers when integrating social services. OpenSocial[10] is a set of standardized APIs for building social applications: its goal is to provide a common interface that allows an application to run on any OpenSocial enabled platform.

As described in [11], OpenSocial applications are based on the gadget architecture developed by Google: a gadget is a XML document containing HTML and Javascript code that is rendered by an OpenSocial container and shown into the social network site. The interaction between the gadget and the container allows the application to access the social graph of the site. This communication is based on Ajax requests that are standardized in the OpenSocial Gadget Specification[12]. In a similar way, OpenSocial Social API Server Specification[13] defines the REST interface for publishing the social network information. Figure 2.4 shows an example of architecture where an OpenSocial Container implementation[2] is used to extend an existing social networking platform.

[2]A reference implementation is provided by the Apache Shinding project.

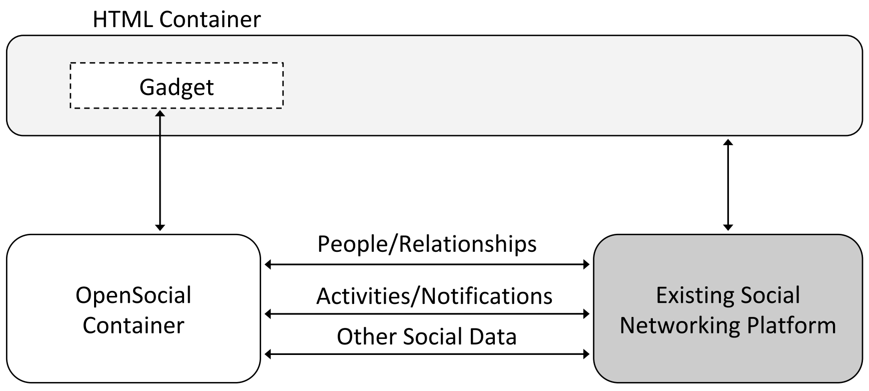


Figure 2.4: OpenSocial Container architecture

The social graph information is designed to be interoperable and independent from the application context of the social networking site. The primary set of data is the one concerning people and their relationships, that are defined respectively in *opensocial.Person* and *opensocial.Group* classes: the first contains all the properties that characterize the user profile, while the latter are collections of Person instances that are used to tag or categorize other users. The Person class supports more than 50 fields that embrace standard contacts information as well as more socially oriented data like interests, skills, and preferences about music, movies or food. As social networking sites may have very different contexts, from platforms based on a specific topic to enterprise solutions, the majority of these profile fields is optional: the specification requires only the presence of a unique identifier (*id*) and a human readable name (*displayName*). More details can be found in the OpenSocial Social Data Specifiction[14].

Besides the information about a user and its relationships, OpenSocial standardizes also the common features that can be found in a social networking site. These include the support to the Activity Stream specification, a dedicated API for private messaging and services for the creation of albums and media contents. Moreover, OpenSocial Containers ease the development of applications by providing a simple persistence layer, which allows developers to store key-value pairs of string for each application and user.

2.2.3 SNeW

The fragmentation of social networking services that we previously described has encouraged attempts for the creation of open standards to allow federation and interoperability among heterogeneous services. SNeW [15] is a specification defined by the Open Mobile Alliance that allows interoperability between social network clients and servers, and federation among social services, so that users can easily communicate with users on other social networking services and migrate their data, when using a different service provider.

Various scenarios of user interactions supported by SNeW-enabled social networks are described in [8]. In the first scenario, a standard-compliant social network is interconnected with external, non standard-compliant services, allowing users to leverage possibly different services with a seamless interaction. In particular, a user posts a status update using the standard compliant service; this update is propagated to all external interconnected services. A friend of the user on an external social network posts a comment to the status update, and the author receives a notification on the standard compliant service.

A second scenario introduces the federation capabilities of standard-compliant services by describing a common social interaction involving three users, A, B and C. A takes a photo using his smartphone and posts it on a social network, explicitly tagging B, who receives a notification. C, who is a friend of B, posts a comment on the photo, and both A and B receive a notification. The innovation brought by standard-compliant social networks is that the three users can use possibly three different services, and that each user only needs one account to use any of these services.

The architecture of a standard compliant social network is shown in picture 2.5. The two main components of the architecture are a SNeW server, which acts as the service provider, and a SNeW client, which is used by users to access the service. The communication interfaces are standardized, thus a client can interact with servers of different services. The standard also provides a set of interfaces allowing the interaction with external applications, both client-side and server-side, and with external, standard compliant social networking services. A SNeW server also implements a gateway component, which enables the interaction with external, non standard-compliant services, by means of connectors that mediate the access to proprietary interfaces.

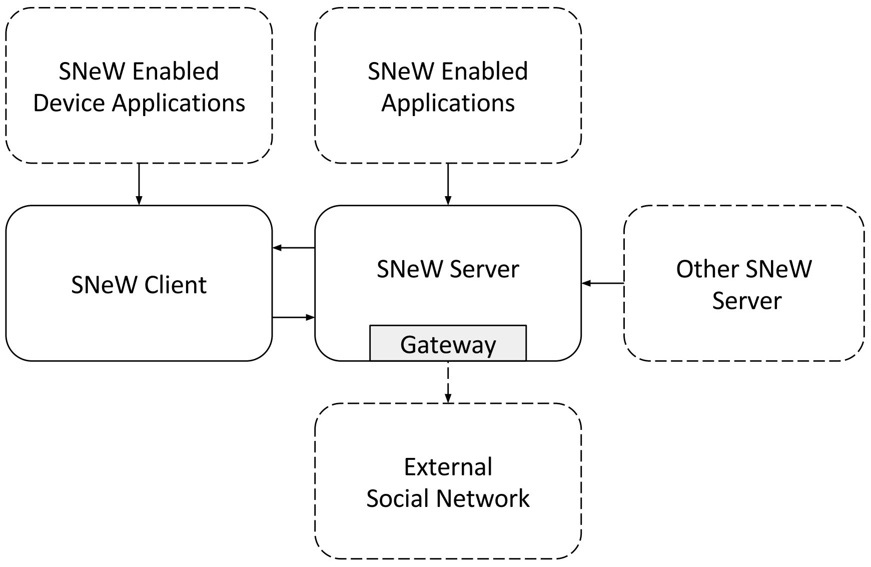


Figure 2.5: SNeW architecture overview

Among the several standards used to specify the interfaces between the various components, OpenSocial is used to define the data format and the interface to access social information.

2.3 Frameworks for proximity social interactions

The steady increase in the adoption of mobile devices, in particular smartphones, has changed the way in which we interact with other people. These devices are capable of accessing the Web by means of cellular network or Wi-Fi, and their operative systems often allow third-party developers to implement full-featured mobile applications by means of SDKs provided for free by device makers. All these factors, together with the new focus on social interactions, contributed to the diffusion of social applications that foster real-life, casual interactions between users in proximity, as described in [16].

As most of the social applications share a similar communication layer, their development can be greatly simplified by middlewares that provide a simpler communication interface tailored to the social context. In this way, developers can focus their efforts on the implementation of user functionalities, without dealing with the complexities of the networking details.

This section describes some notable examples of social proximity middlewares, developed in academic contexts. These examples can be divided into three categories:

- Middlewares that apply semantic techniques to extract social relationships from available profile data.
- Middleware that aim at extending existing web social networks into the mobile world.
- Middlewares that provide support for resource sharing.

2.3.1 MobiClique

MobiClique [17] is a mobile social networking middleware that allows users to maintain and extend their connections in the virtual world as they meet new people in the real world. It is designed to be an extension of existing social networks services into the mobile world: in fact, the set of existing relationship of a user is obtained from its user profile on an external service during a bootstrap process.

In its prototype form, MobiClique does not embrace any standard for the representation of the user profile, as the developers chose to rely on a simple representation comprising a unique identifier, a user name, a short description, a list of friends and of interest groups. However, according to the developers, the profile representation can be easily replaced with an existing standard, like OpenSocial. In the prototype, the MobiClique profile is bootstrapped using the Facebook API to read the value for each profile field. In particular, the list of friends is used by the middleware to build a *social graph*, which is then enriched with encountered MobiClique instances and their friendship relationships.

Each MobiClique instance executes a periodic loop composed of three steps: *neighborhood discovery*, *user identification* and *data exchange*. When a new device is discovered in proximity, MobiClique enters the identification phase, in which the two devices open a communication channel to exchange profile information. Upon the first encounter, the profile is transmitted completely and stored persistently; during subsequent contacts, the profile is transmitted again only if it has changed since the last encounter. After the identification is completed, the devices can exchange application level messages which are stored persistently on the devices if enough storage space is available.

Opportunistic exchanges combined with user mobility allow message to travel from device to device over multiple hops, without any infrastructure, until they reach their destination. Messages can be addressed either to a single user or to an interest group: messages directed to a single instance are directly delivered if the destination is in proximity, otherwise the middleware finds all the paths from the local user to the destination on the social graph, and forwards the message to the first user of each paths upon contact; group messages are flooded in a similar way to all users within the group. All messages are characterized by a Time To Live (TTL), after which they are automatically deleted by the system.

The MobiClique prototype does not implement any privacy settings and thus users cannot limit the visibility of their profile information, which is a major drawback for the platform. Moreover, as the middleware employs Bluetooth networking for communication, MobiClique will incur in scalability and efficiency problems.

2.3.2 MobiSoc

MobiSoC [18] is a social computing middleware that improves the development and deployment of mobile social applications by providing developers with a series of capabilities: first, the platform provides a mechanism to capture dynamic ties between users and between users and places. Those ties are then modelled, validated, stored and made available to a multitude of applications. Secondly, the platform provides a centralized infrastructure that enables the efficient and scalable collection of user locations in real time. Lastly, data collected by the centralized infrastructure is used to create a model of the global state of the community, in order to identify emerging geo-social patterns.

MobiSoC focuses on capturing and managing the social state of physical communities by analysing collected social data. The state of a community is represented by a collection of user profiles, place profiles, social relationships between users and associations among users and places. This social information is continuously evolving as new people, places, relationships and associations are created. Moreover, learning algorithms can discover new relationships between entities by analysing geo-social patterns; this information is used to improve user and places profiles and to suggest new relationships to end-users.

MobiSoc is based on a centralized service that implements all the data collection and learning capabilities, while mobile devices interacts with the public API of this service by means of a thin mobile client.

This approach has multiple benefits: first, the most computation-intensive part of the application is executed on the server, resulting in a more energy efficient mobile application. Secondly, a centralized entity allows to easily maintain a consistent view of the social state; in a decentralized configuration, the social state would be split on multiple devices, thus limiting the data available to learning algorithms. Lastly, a centralized storage mechanism allows a persistent storage of profile data and a better access control management.

The choice for a centralized architecture, however, has some important drawbacks that may prevent the platform from gaining popularity among developers. In fact, the centralized entity is a single point of failure for all mobile applications powered by MobiSoc, as they would stop working during server downtimes or when a network connection is not available.

2.3.3 Mobisoft

Mobisoft [19] is an agent-based proximity middleware that enables the creation of peer-to-peer overlays on top of mobile ad-hoc networks. The middleware focuses on the concept of *Personal Area Network*, as the digital space around the user device that can be reached using a wireless communication technique; in particular, Mobisoft uses Bluetooth for its proximity communication. When two personal spaces overlap, users can virtually "see" each other, i.e., their mobile devices are able to communicate and exchange information. With these assumptions, there is no need for an explicit notion of *space*, as it is an inherent network abstraction.

The introduction of software agents improves the autonomy of the middleware by reducing the need of user intervention for common tasks. In the Mobisoft middleware, agents are tasked with supporting the natural human behavior of information exchanging by searching for adequate communication partners and interesting information within the user's Personal Area Network. Once a partner has been discovered, involved agents automatically start information exchange by transmitting semantically annotated messages; only this first step is transparent to users to negotiate the best interaction time. Subsequently, users are notified of the discovered communication partner and can decide to take the further steps.

Each user within the Mobisoft middleware is characterized by a *profile*, which is a collection of meta-data saved persistently using the Resource Description Framework (RDF), together with the Friend Of A Friend (FOAF) vocabulary; the data model consists of *resources* and *statements*, which link together two resources using a predicate, in a subject-verb-object configuration. This model allows the description of common personal data of a user, and also the list of people he knows; to describe the list of interests and preferences, custom elements have been added to the standard FOAF vocabulary. Thanks to the adoption of RDF, agents are able to semantically analyze the profile information received from another user upon meeting to determine whether his profile is relevant according to the information contained in the local user profile. If the two profiles are semantically relevant, the middleware notifies the user as previously described. Whenever a foreign agent gains access to sensitive profile data, the user is notified and requested for acknowledgment.

2.3.4 MyNet

MyNet [20] is a social networking platform that allows users to share their personal resources, devices and services in real time over a secure peer-to-peer network. The platform does not depend on a centralized infrastructure and is designed to be usable also by users without technical expertise. The basic building block of MyNet is a *device*, defined as a routable and authenticable overlay network endpoint, uniquely identified by an Endpoint IDentifier (EID). A new device is integrated into the platform by means of an *imprinting* process, which imprints the owner identity, profile and secret (e.g., a PIN) onto the device. The secret is used to authenticate the user, and can also be used to protect a configurable set of platform functionalities.

The set of devices within the MyNet platform that belongs to the same user is called *Personal Device Cluster*. After his identity has been imprinted onto a device, the user can add it to its PDC by means of a *device introduction process* that merges the new device with one already in the cluster. The process requires authentication on both the devices and is reversible.

Users can establish social relationships by linking together their clusters by means of an *people introduction process*, which requires both users' approval and results in the addition of a new social contact to their PDCs; once the relationship is established, users can share their resources with he other at any time. During the linking phase, the P2P layers of the involved instances exchange routing information and the identifier of each instance to allow future communications. Users may also create *groups* of users or devices for a fine-grained access control; in particular, user groups can be used as the recipients of an access control privilege, while device groups as targets. For ephemeral sharing scenarios, MyNet provides *Passlets*, a metaphor for real words "passes", which can be used to grant temporary access to a resource within the user's PDC.

Each PDC can host one ore more *user-accessible services*, i.e., services perceived by the user, like chat or picture-sharing; each user service is implemented by one ore more elementary components generally transparent to the user, which run on one or more devices of the user's PDC. In particular, services can be divided into *MyNet-aware*, which are services implemented using the public API of MyNet, *MyNet-enabled*, legacy services for which explicit support has been implemented into MyNet, and *MyNet-transparent*, legacy services of which MyNet is unaware. Services can be used to implement a broad set of capabilities, including file sharing: in the general case of resource sharing services, MyNet delegates privacy management and access control to the service itself.

2.3.5 Samoa

Samoa [21] is a middleware developed at the University of Bologna that supports the creation of semantic, context-aware and roaming social networks. It provides a logical abstraction that groups together users in physical proximity who share affinities and interests. Mobile users can play three roles within the framework: *manager*, which is the creator and owner of a social network; he is responsible of the definition of its discovery scope and of the criteria that new members should match; *client*, a user which is located within the discovery scope of a social network and is eligible to become a member; *member*, a user which is already affiliated with the social network. A single user may play multiple roles at the same time in the context of different social networks. The management of a social network is based on the concept of *space*, which is the physical space within its discovery scope: the manager is the centre of the place, while clients and members are users whose distance from the centre is less then a given number of network hops, defined as the *place radius*.

All Samoa entities, i.e., users and places, are described by a profile, which is a set of meta-data grouped in different categories according to their logical meaning. A *place profile* is characterized by an *identification* part (comprising its identifier, name, description…) and an *activity* part, listing the social activities and interaction supported by the place that users can share. A *user profile*, on the other hand, features an *identification* part, comprising his personal details, together with a *preference* part, describing the social activities he is interested in. A third type of profile, the *discovery profile*, allows the manager to define the preferences that a client must declare to be eligible to join one of his social networks.

The Samoa framework supports two types of social networks: *place dependant*, which only includes the members currently co-located with the manager, and *global*, which comprises the whole set of place dependant social networks created over time by the managers. Samoa automatically defines place dependant social network leveraging two semantic-matching algorithm. The first matches all the profiles of clients in the discovery scope against the place profile: those users whose profile contains preferences semantically related to the place's activities are eligible to become members. The second algorithm takes as input the set of eligible clients and selects those with profile preferences which are semantically related to the place's discovery profile: the profiles of new members are saved in the place-dependant social network, thus incrementally building the global one.

Regarding user privacy, the Samoa middleware does not offer any access control capability, and thus it is impossible for users to limit the visibility of their profile information.

2.3.6 Yarta

Yarta [22] is a middleware platform that supports the development of mobile social applications focused on the analysis of the social graph of users. It allows applications to share and reuse their respective knowledge by using an expressive and extensive model to represent mobile social environments and their contextual interactions. The data model of Yarta is based on the Resource Description Framework (RDF) and uses a vocabulary inspired by Friend Of A Friend (FOAF); in particular, the base model has been enriched with additional elements to support the complete set of the functionalities of Yarta. Social data is stored and managed by a *knowledge base* which offers a set of API to access, update and remove social data. User applications are able to retrieve data from the KB by means of high-level queries that hide the internal RDF representation; queries may also be executed on the KB of a remote user.

Data stored in the KB is organized in an uniform graph of social information: the initial node is the local user, and additional nodes are included when new relationships are discovered by means of *social sensors*. These sensors are components provided by Yarta that are able to populate the KB by automatically extracting social informations from two categories of sources: the first category comprises all those sources that natively support the concept of *relationship*, for example existing social network services; for these sources, sensors can be implemented using the external service's API to import data into Yarta. The second category includes sources that contain social information without native support for relationships, like the call log on a mobile phone; for this sources, sensors must employ inference algorithms to guess existing social relationships from available data.

Yarta natively provides a flexible access control that can be finely tuned by the user according to his social preferences. It employs semantic policies represented using SPARQL, a query language for the retrieval and manipulation of data stored using the RDF format. Policies are managed, evaluated and enforced by the *Policy Manager*, a component decoupled from both the application logic and the KB management. The Policy Manager intercepts any read, add or remove operation on elements of the KB and performs reasoning on the defined policies and the context of the request to determine whether the action should be permitted.

Yarta is implemented with a decentralized architecture and supports multiple communication technologies by means of a multi-platform communication layer that provides both synchronous and asynchronous messaging, a data transfer mechanism and a network-agnostic naming and discovery service that allows to detect Yarta instances in proximity.

2.4 Conclusions

Recent years have seen the birth of emerging technologies that prove the interest of industry about device to device communication. This enthusiasm is driven by the novel IoT scenarios and the widespread diffusion of mobile devices equipped with more and more powerful capabilities. In particular, the availability of powerful hardware and several networking technologies enables the development of new sophisticated services, changing the way in which we interact with the physical environments.

Most of these novel technologies focus on dealing with the heterogeneity and details of proximity networking, thus providing essential tools to create proximity based applications. Some of them provide even higher level of abstraction, like Alljoyn and its RMI model, that speed up the development of such applications. However, none of these fit properly the constraints that modern mobile application development impose on the life-cycle of internal components. Thus, a developer must handle and build from scratch the whole infrastructure needed to manage the lifespan of published services; this includes: the management of connections, multi-threading and synchronization issues, and the complexity of implementing background components.

Existing solutions completely neglect the social element, which was fundamental for the rise of the modern web. In fact when defining and publishing proximity services, user identity is often seen as accidental, thus resulting in a meaningless mash of device and service names. Still, social aspects can provide additional value also in a proximity context, by opening new interesting scenarios and allowing applications to offer services that are tailored to their users.

In the academic world, multiple research teams have tried to develop middlewares for social applications in proximity environments. These social middlewares, built on top of proximity communication technologies (Bluetooth or Wi-Fi), provide social capabilities to support real-life interactions among users, including semantic profile matching, resource sharing and notification of people in proximity. However, most of these works lack of a thorough analysis of the differences between web scenarios and those of mobile proximity environments, thus they do not embrace all the novel opportunities offered by the mobile environment. As result, the capabilities of modern mobile devices are under-evaluated and not fully exploited, and the proposed solutions present limited applicability: a set of features that are borrowed from already established web social networks, without providing the building blocks for more general scenarios. For example, all these middlewares lack facilities to allow developers to quickly implement custom interactions, beyond those supported by default, although they are enabled by modern networking solutions.

This thesis work wants to propose an infrastructure for the creation of novel social proximity services, by providing a framework that unifies the capabilities of network technologies and the potentiality of the social features. The final aim is the creation of an environment in which the user identity is once again brought into focus and where different applications can coexist and integrate themselves with seamless interaction and full control on behalf of the user.

Chapter 3

Social Proximity Framework

This chapter presents the Social Proximity Framework (SPF), topic of this thesis. SPF is a software solution for the creation of *social smart spaces*, where social identity is taken into account as a key enabler to provide services more tailored to users and to enhance real interactions between people with digital ones. SPF offers a thorough infrastructure to handle the development and deployment of these social proximity services.

Section 3.1 introduces the most significant application scenarios and outlines some high-level requirements for SPF. Here we provide a definition of social proximity application and social smart space.

Section 3.2 presents the tools that SPF offers to developers for the creation of social proximity applications. These tools include a shared user profile, utilities to implement social proximity services, and two different means for the discovery of remote users.

Section 3.3 discusses the global vision of the SPF infrastructure, in particular describes how applications and personal data are managed within the framework. The concept of SPF Provider is introduced as a software module that provides a user interface to control personal data and services held by the framework.

3.1 Problem analysis

As discussed in Chapter 2, the emerging of several new technologies in the field of the Internet of Things is changing the way in which we interact with the environment and its objects. Device to device communication is seen as a promising solution to avoid the need of internet connectivity and to fit the requirements of battery-powered mobile devices. In practice, actual applications of these new solutions can be found in the proximal discovery of services and entities, as in the case of Bluetooth LE beacons, and in the remote control of smart objects.

What is missing in this scenario is the social identity of the user, which is usually neglected or, if considered, is based on external infrastructures. This solution has limited applicability in different scenarios, for example in absence of connectivity or in public environments where there is the need to deal with heterogeneous devices and social services. In these cases we think that a unified software layer and device-to-device communication can help in building smart spaces that are focused on the identity of their users and offer novel social proximity services.

A *social proximity application* is not intended to be only a replica of existing web-based social functions. Rather, its aim is to support daily activity of a smart space and to offer services tailored to the users, on the basis of interest, skills, and other information of their profile.

The union of different social proximity applications should create a new kind of smart space with a social flavor, as usually happens almost everywhere in the web. These novel social smart spaces offer means to discover services, people, appliances and to support the communication between them. The purpose of such a smart place is not to replace real social interactions, but to encourage them through digital ones.

SPF is a software solution that aims at creating social smart spaces, providing advantages for both developers and users. The framework offers to the developers the advantage of ready to use facilities for the implementation of social proximity interactions. Users and smart space owners can use the framework to control their information and manage the installed services.

The following section discusses in detail the concept of social proximity application, providing some of the most meaningful application scenarios.

3.1.1 Application Scenarios

Social Network in Proximity

The Social Network in Proximity (SNiP) scenario aims at describing how standard social networking functions can be adapted to a proximity context. A recent trend for social networking applications is utilizing proximal discovery for social matching: proximity is seen as an important factor to determine the relevancy of a match. Device to device communication can be exploited to avoid the need of GPS location tracking and external infrastructure. A P2P architecture that does not require a centralized solution, may help in providing the social functions of the SNiP even in environments where there is neither Wi-Fi access nor internet connectivity.

Such functions are intended to stimulate real life interactions by means of digital ones; this is especially true in large events, crowded of people. In a career day, people advertise their skills and companies discover interesting profiles. In a fair, customers interested in particular brands or products can discovered their representatives. In a party, people can meet new people on the basis of common interests. Each user of this service has to:

- create a social profile;
- add a username and a profile picture;
- add interests (e.g., cinema, Japanese food, soccer);
- publish the social profile in the smart social space.

A user then can use the application to discover profiles according to personalized queries. Once the interesting people are discovered, the user can communicate with them by sharing messages and contents as well as by means of different types of communication offered by the application. Such interactions may resemble the ones that can be found in social networking sites or even be specifically designed by the application.

Targeted Advertising

This scenario describes the provisioning of tailored digital advertisement, like discounts, promotions or special offers, to customers of shops, restaurants and bars in physical proximity. This service, deployed on special devices physically placed in these shops, is able to interact with customer devices that have a specific application installed.

Shop owners can configure the service by creating new coupons and special offers for customers. The provision behavior of the service can also be configured, for example to deliver a coupon when a user enters the shop for the first time, or provide additional promotions to loyal customers. On the other hand, the customer application allows users to specify their interests in advertisements for specific topics. Once the customer is in the proximity of a shop, the service reads the interest list from his device and provides him advertisement that matches his interests.

As an example, customers may leverage this application to declare their interest in coupons and offers related to smartphones. Owners of shops and pubs can set up the service to discover customers in proximity interested in the products they sell, in order to perform targeted marketing campaigns. Upon launch, the service performs multiple, periodic searches to discover enabled devices in proximity. Once a new device is found, the service provides a coupon according to its configured behavior; the user is notified of the new coupon by means of a push notification received on his device.

The idea of proximity-based advertisement is not new. Recent solutions, including iBeacons or other indoor positioning systems, have been installed inside shops and restaurants to discover the presence of customers within stores, and provide them advertising. However, these solutions traditionally lack a built-in support for user profiling, and thus require an external, ad-hoc infrastructure, and an internet connection for intercommunication.

On the contrary, in this scenario coupons and special offers are actively advertised to potential customers located in the proximity of the shops. Moreover, the advertisements are tailored on user profiles, thus improving the effectiveness of advertising, without depending on internet access or an external infrastructure.

3.1.2 Requirements

On the basis of the scenarios previously described, we have identified the following high-level requirements that a novel framework needs to satisfy in order to support social interaction in proximity.

1. *Profile Management*
   The framework must enable the creation of a persistent and centralized social profile stored on the user's device. Such profile should contain an identifier for the user, unique within the SPF environment, and allow the user to insert his personal details. The profile should be accessible to local applications and remote instances, according to the privacy preferences of the user. Local applications may be allowed to both read and write data on the profile, while remote applications may only read profile information.

2. *Advertising*
   The framework must enable users to advertise their profile to other instances in proximity. Advertising should be performed in accordance to the privacy settings defined by the user; in particular, the user may restrict the set of profile information advertised to peers in proximity. The framework should also allow local applications to react to incoming profile advertisement.

3. *Discovery*
   The framework must support profile discovery by instances in proximity on behalf of local applications. The discovery process should be based on a query mechanism to allow filtering profiles on the basis of profile information. A local application that started a search process should be notified when a new matching instance is found. Discovery can either be fully peer-to-peer, or mediated by a centralized device installed in the smart space.

4. *Communication*
   The framework must provide high-level communication facilities supporting the interaction between user applications in social smart spaces. These facilities should hide the underlying communication protocols adopted by the framework, so that remote applications can communicate transparently, without dealing with networking details. The framework should also provide special support for well-known social activities, such as messages and media-content posts.

5. *Security*
   The framework must allow users to control how their resources

and profile information are shared with local applications and remote instances. In particular, the framework must provide a facility to restrict the access to the profile information only to allowed people and applications.

3.2 Tools for social proximity applications

This section describes the building blocks provided by SPF to ease the development of mobile social applications. First, applications can leverage on a shared *user profile* maintained by SPF, that comprises both user-provided information and contributes from SPF-enabled applications installed on the user's device. SPF also allows applications to register and execute proximity *services*, which enable the implementation of ad-hoc interaction mechanisms among applications.

Another fundamental building block is proximity awareness: applications can discover other SPF instances in physical proximity by means of *search* queries that can be configured according to applications needs. Reactive actions are also supported, as applications may require to be notified of people in the proximity of the user. This is based on profile *advertisement* and applications-defined rules, named *triggers*.

Lastly, SPF introduces the concept of *activities*, a data structure to represent social actions between users, and provides a series of facilities whose aim is to ease the implementation and integration of well-known social interactions.

3.2.1 User profiling

SPF allows users to create a persistent personal profile to represent them in virtual social interactions mediated by the middleware. The profile is managed by the framework on behalf of the user, and provides access to local applications and remote users according to the user's privacy preferences.

The information contained in the user profile can be inserted either directly by the user, or by external applications installed on the user's device. In particular, the content provided by external applications contributes to the creation of a dynamic profile that is automatically updated to best match the real details of a user. For example, a media player can contribute up-to-date information regarding the user's music tastes.

The user profile has a modular structure, organized into a collection of *profile fields* and stored according to a key-value pattern; some of the available fields are listed in Table 3.1. This information model is inspired by the one of OpenSocial, and covers both personal information and additional social data. The modularity of the structure allows for new fields to be easily added.

Field name	**Field type**	**Description**
`Identifier`	String	Unique identifier for a user.
`Display name`	String	The name of this user, suitable for display to other users.
`Photo`	Binary	A photo of this user.
`Birthday`	Date	The birthday of this user.
`About me`	String	A general statement about the user.
`Emails`	String	E-mail addresses for this user.
`Location`	String	The user's place of residence.
`Status`	String	The user's current network status
`Gender`	String	The gender of this user.
`Interests`	String	The user's interests, hobbies or passions.

Table 3.1: Some of the profile fields supported by SPF

SPF also introduces a mean to diversify the profile information, so that applications belonging to different domains are provided appropriate values for each profile field. This diversification capability goes under the name of profile *Personas*: a Persona is a named container that can hold a value for each profile field. A user can create multiple personas, and assign them the most appropriate profile field values. Each application that interacts with SPF is linked with a specific persona, as discussed in Section 3.3.1, and can only access the field values contained in it.

In this way, users can create multiple user profiles, each one tailored to a specific context. For example, a user can create a "Business" persona to hold work-oriented details, and a “Personal” one to hold informal details for friends and family. The business persona shows his business email address and his work skills and interests; the personal one, on the other hand, provides his personal email address, and his favorite hobbies and brands as interests.

3.2.2 Services

The implementation of rich interactions between applications installed on different devices is a complex and time-consuming task for application developers. In fact, developers need to rely either on a networking technology for direct communication, with the burden of low level details, or on an external cloud infrastructure, increasing costs and development time.

As discussed in Chapter 2, a solution to this problem is a middleware that provides high-level abstractions for device to device communication. These abstractions allow the developers to avoid both the development of an external infrastructure and the intricacies of low level networking details.

SPF tackles these issues by providing facilities that allow applications to define custom software functions called *services*. SPF enabled applications can seamlessly execute services of external applications, either installed on the local device or on a remote one.

The capabilities offered by a service are declared by means of an *interface* that lists all the methods available for invocation; each method is identified by a name, which must be unique within the service. Methods may accept input parameters, provided by applications upon invocation, and a return value that is provided back when invocation is concluded. The service interface also provides a set of meta-data describing the service, including its name and version. The business logic that realizes the interface is provided by means of a service *implementation*, which contains the code to react to a service invocation.

SPF provides a Remote Method Invocation mechanism that enables the remote execution of application services. The central part of this mechanism is the *Service Registry*, which stores all the services that are available for execution in a SPF instance. To make a service available to others, an application first needs to register it in SPF, which stores the service’s meta-data in the registry.

The invocation of a remote service happens transparently by means of a *Service Stub*, a component that exposes the same interface of a service, hiding all the lower-level communication details. To obtain the stub for a service, an application must first identify which service wants to invoke by providing the identifier of the service to invoke, and the SPF instance that is exposing it.

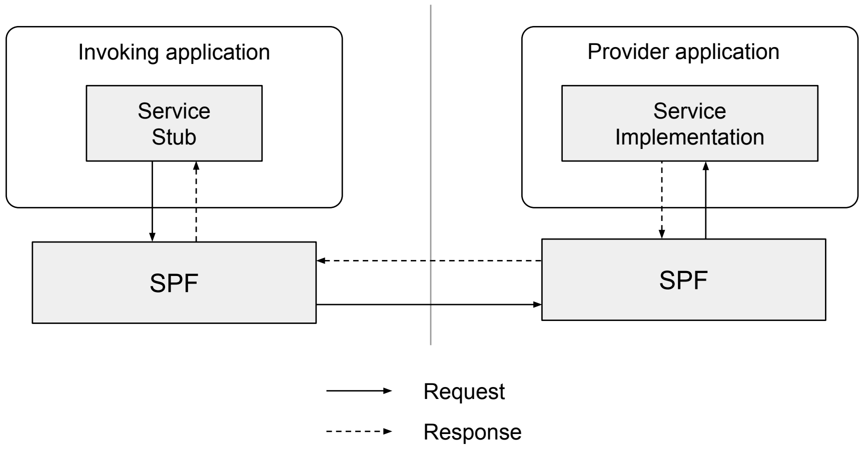


Figure 3.1: Invocation of a remote service

Once the stub is obtained, the application can invoke a method of the service, as shown in Figure 3.1. To invoke a method, the application has to provide the method name and the parameters, if required, to the stub. The serialization and de-serialization of the parameters are handled transparently by the invocation stub. When the stub receives an invocation request, it creates a package containing the meta-data required to identify the method to invoke, together with the serialized parameters; then, SPF dispatches the package to the remote application.

When SPF receives an invocation request, the service is looked up in the Service Registry. If the service is found, the method implementation is retrieved and executed, providing the arguments if needed. Once the service has been executed, a response is sent back to the application that invoked the service; the response contains a status to signal any error, and the serialized return value, according to the method declaration.

3.2.3 Search

The discovery of other people located in physical proximity with a user is a core feature provided by social proximity middlewares, as discussed in Chapter 2. SPF provides a flexible and configurable search service that enables the discovery of other SPF instances that are reachable by means of the underlying networking technology.

SPF supports a complete customization of the search process. In particular, an application can restrict the set of discovery results by means of a query that states a series of properties that results must match. A search property can be of three types:

- **Field value**: states that a profile field must have a specific value. For example, a query may require that the “gender” profile field is set to “female”.
- **Tag**: specifies a values that must be contained at least in a profile field. For example, the value “Android” can be contained in the “interests” profile field, as well as in “skills”.
- **Application**: specifies the identifier of an application that must be registered in SPF.

A query matches a person if all the specified query properties are satisfied by the instance. For example, a query to discover all the women related to a tag “Android” should specify “female” as value of gender profile field, and “Android” as a tag.

Search queries are executed by means of a distributed mechanism that delegates the query evaluation to remote instances. As shown in figure 3.2, queries are broadcast to all reachable SPF instances by means of search signals. Upon reception, queries are evaluated against the user profile. The search signal also contains the identifier of the application that requested the discovery process; if the application is installed on the local device, SPF reads the profile value from the persona that is assigned to that application, otherwise from the default one.

If all the properties match, the query is fulfilled, and a search result signal is sent back to the instance in discovery. The search result signal contains also the basic information of the matching instance, such as the user identifier and the name to be displayed. This information is

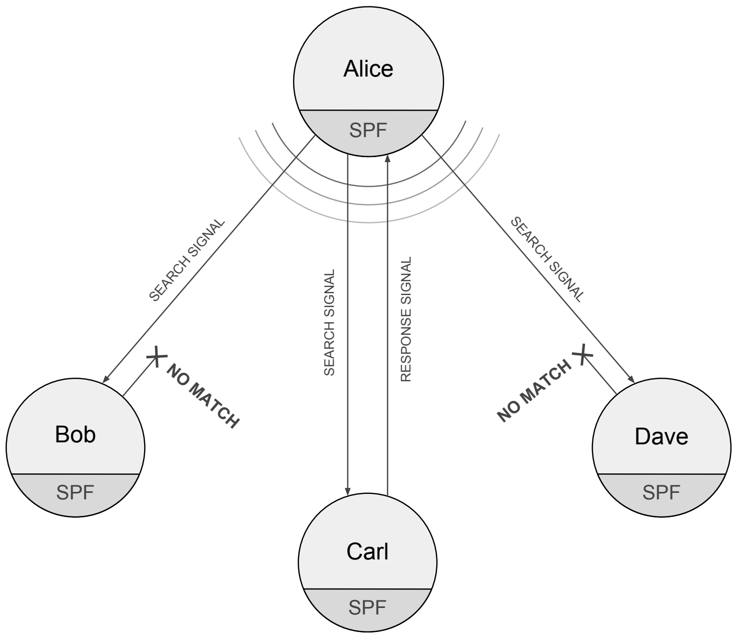


Figure 3.2: Execution of a search request

provided to the application that started the discovery request to avoid additional requests to retrieve this information.

To start a discovery process, an application must provide a query, some configuration parameters and a callback to be notified of results. The configuration enables applications to control the number of search signals sent and the time interval between two subsequent signals. The callback, on the other hand, allows SPF to dispatch search events to the application that started the discovery process. In particular, the application is notified of the availability of a new user that matches the specified query, and when a previously notified user is no more available. In this way, applications can progressively build a list of people available in proximity. Applications are also notified of the status of the search process, in particular when the search starts, when it stops after all signals have been sent, and when an error occurs during the search.

3.2.4 Advertisement

Besides the search feature previously discussed, SPF offers another mechanism to discover remote instances. This functionality, called *advertisement*, allows devices to exchange profile data, without user intervention.

In analogy with the presentation between two people in real life, advertising is performed by exchanging profile information when two or more devices become in proximity of each other. This profile information comprises the values of a set of profile fields, and possibly the list of installed applications. The user can configure this function by selecting which fields to advertise, whether or not to include the list of applications, and the persona from which these pieces of information are retrieved. Moreover, it is possible to decide whether or not to turn on this feature: once enabled, it is tied to the network resources of the framework, meaning it works as long as SPF is active. This, along with the control over the exchanged information, should ensure enough privacy and adequacy to different contexts of use.

For example, a user may want to advertise business related information while working, and turn off the feature when in his spare time. Differently, a device situated in a shop may want to continuously advertise itself with the purpose of attracting interested costumers.

Application can react to received advertisement by defining *triggers*: a trigger is a rule that specifies an action to perform when the received advertisement satisfies a given condition. Conditions are specified by means of search queries, as the one available for the search, while actions can be of two different types:

- the *Send Message* action delivers a textual notification to the remote device that has activated the trigger;
- the *Intent* action notifies the application that has defined the trigger about the remote instance.

Figure 3.3 describes how triggers are handled: upon the reception of an advertisement, data are evaluated against all the registered triggers; in the case that a match is found, the action associated with the trigger is executed.

SPF allows developers to configure timing parameters that modify the behavior of a trigger. In detail, it defines a *sleep period*, that is the time interval during which subsequent activations of a trigger on the same SPF instance are blocked. If triggers do not specify a sleep period, they are automatically configured as *one shot*, meaning that their actions can be executed only once for each remote user.

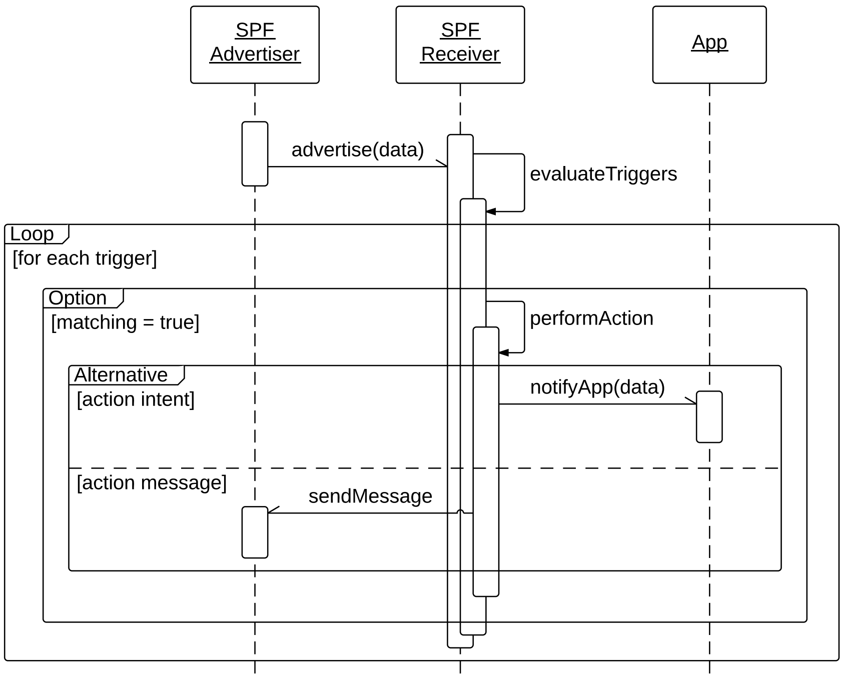


Figure 3.3: Sequence diagram: trigger evaluation and action execution

In short, the triggers mechanism allows applications to define persistent queries that are fully managed by the framework in a transparent manner. This feature targets specific use cases where there is the need to provide an always-on search. To obtain a similar behavior with the standard search, a developer has to deal with its own background components; instead with triggers there is a shared background component that makes the development of such a function much easier. Moreover, it offers the opportunity to optimize the use of resources, especially when multiple applications have such similar behavior.

Triggers are a flexible solution, that well suits the targeted advertising scenario discussed in Section 3.1.1. Here follow two concrete examples of how triggers can be used in such scenario. We distinguish two main actors: the first is the entity that wants to attract new potential users by letting them discover its information, e.g, a shop; the second is the user that is the object of the interaction. For simplicity, we call these two actors respectively *advertiser* and *customer*.

In the first interaction pattern, customers advertise their own profile by means of SPF. An application installed on the advertiser device can then define triggers for executing actions that are tailored to the customer. These actions can include the sending of a simple message or the execution of a proximity service defined by an application installed on the customer's device. The latter can be easily implemented through an intent action provided by the SPF triggers.

The other pattern is the symmetrical counterpart of the first one. Here the advertiser uses SPF to broadcast its profile information. Customers' applications can define triggers to notify the presence of an interesting entity. To have a more concrete example, this kind of interaction can be used in shopping centers where there are multiple advertiser and a large number of customers.

3.2.5 Activities

Mobile social applications offer a set of well-known services that support social interactions among users. Often, the same service is offered by multiple applications in a similar way; this is particularly evident in the case of mobile chat applications, where multiple, widely-used options are available. However, even if the provided capabilities are similar, these applications are almost never inter-operable, thus the interaction among users of different services is not possible. In this scenario, any generic interaction, like a chat message, requires the sender and the receiver to have installed the same application, as result this is a limitation for the diffusion of new applications.

Even though SPF services enable the interaction among different applications, they are not enough to solve this interoperability issue. In fact, multiple similar interfaces can be defined to model the same interaction, thus requiring the adoption of a standard. The same standard interface, though, can be exposed by multiple applications on the same device, thus developer would still need to specify which application they want to interact with.

To solve this problem, SPF introduces the concept of *Activity*, inspired from the ActivityStream [9] specification. An activity is an inter-operable key-value data structure, containing standardized pieces of information that characterize most of the common social interactions. Each activity defines common attributes that identify the subject of the action, its target, a verb that specifies the type of action, and other attributes that provide additional information, such as date and time. Besides these common data, an activity contains attributes that constitute the content of the action they represent, for example a status update may include a textual message or a picture, while a check-in should include a location.

Activities can be created locally on the device or can be sent to remote users when the action includes the participation of other people. This is the case of writing a post on a friend's wall or even sending a chat message. All these interactions can be handled by the framework without dealing with specific interfaces of SPF services.

SPF offers an activity routing mechanism based on the verb attribute, that is the type of activity. SPF services may leverage this feature by declaring themselves as activity consumer for certain verbs. From the client point of view, sending an activity does not imply the knowledge of any interface of application. The framework takes care of dispatching the activity on the basis of its registered services. To give a more concrete example, consider two different chat applications; if they want to adopt the activity routing mechanism to implement the exchange of textual messages, they only have to agree on the verb of the activity and then extract contents and identity information from the provided data structure. Moreover, when using activities, the framework automatically adds some pieces of information, including the date, the creation time, and details about the user identity such as his identifier and name, thus avoiding useless additional readings from the profile.

In conclusion, SPF activities provide a facility that eases the integration of standard social interactions into social proximity applications. For example, integrating a chat functionality is as simple as handling activities with the verb "chat", by displaying incoming messages to the user and sending outgoing ones to remote users. Activities are built upon services, but are not intended to replace them: in fact, services are required to implement custom interactive behaviors, like proximity-based games or applications that support the specific business of a smart place.

3.3 Infrastructure

As described in the previous section, SPF allows users to store their personal information along with services registered on behalf of installed applications. These resources are then published and made available to interested consumers, both remote users and locally installed applications.

The framework offers different ways for users to monitor the resources and to control how such information is shared. These control functions can be accessed by the user through a *SPF provider*, an external application that offers the interface to manage the framework settings and data.

This section discusses how profile data, services and applications are managed within the framework. First, we discuss how applications register themselves in SPF: this is a delicate process in which the user is made aware of the app intentions. Second we describe the access control model that the framework adopts to manage the access of remote users. Last, we provide an overview of the functions a user can access on the framework to control and personalize his services.

3.3.1 Management of SPF-enabled Applications

SPF allows developers to create proximity-aware applications by means of its public APIs. The functions offered by SPF include access to profile information, search of people in the proximity, registration of executable services and triggers. To prevent the abuse of these functions, the framework focuses its attention on two main aspects:

- SPF notifies the user about applications that want to interact with the framework and allows him to decide whether or not to grant the authorization.
- SPF gives the user full control on the personal information that is provided to applications.

These concerns are addressed by offering an authorization mechanism inspired by the OAuth protocol, and by assigning a different persona to each application in order to differentiate the provided information.

To access SPF functions, applications have to request a permission for each component they want to use. Permissions are used to control what applications can do and require the authorization of the user. Examples are the permission to read from the user profile, the one to register spf services, and to search for remote users. Table 3.2 shows some of the permissions that can be requested by the applications.

Permission	**Description**
Read user profile	Grants read access to the user profile.
Write user profile	Grants write access to the user profile.
Read remote profile	Allows the application to read the profile of remote users.
Register services	Allows the application to define custom proximity services.
Execute remote services	Authorizes the application to execute spf services of remote users.
Search remote users	Allows the application to access the search functions of SPF.
Notification service	Allows the application to define SPF triggers.
Define activities	Allows the application to define a spf service as an activity consumer.

Table 3.2: Some of the SPF permissions

The authorization process is completely transparent to the developer and starts when the application interacts with the framework for the first time. The first step regards the authentication: each application has to provide a unique identifier along with the list of permissions to request. Then the framework validates the actual installation of the app on the system and retrieves the data, such as icon and app name, to be shown to the user. Once the user grants the permissions to access SPF, the framework assigns a token that can be used in the subsequent requests to the APIs.

From the user's point of view, the authorization implies the acceptance of a pop-up with which he is made aware of the permissions that will be granted to the app. The pop-up allows the user to select the persona of the profile to be associated with the app and provides additional information in order to let him recognize the application which is asking for the authorization.

3.3.2 Remote access control

The main objective of SPF is to enable proximity based communication with other people: this communication includes the access to personal information of the profile and the execution of services. Such information may not be intended to be shared with everybody at the same level: what the user desires to share with other people depends on the level of familiarity. Therefore, to allow full control on how the information is shared, SPF offers a *contact request* mechanism that allows the user to manage the access to his resources. This mechanism provides a clearance model based on the concept of *groups*. Users can use groups to categorize their contacts and restrict access to specific resources.

Due to the proximity nature of SPF, the process has been designed to be as short as possible; it is based on a single message exchanged by the two involved SPF instances. When a user wants to send a contact request to someone, a dialog asks for confirmation and allows the user to select the groups for the new contact. Then the local SPF instance generates a token associated with the identifier of the receiver and stores it in a table containing the list of contacts authorized to access the user resources. This token is sent to the remote instance in a contact message, together with some identifying information of the sender, such as the profile picture and the name. Upon reception, the information contained in the message, along with the token, is stored by the remote instance that notifies the user about the pending contact request.

The user can decide whether to accept the request or not, and select the groups for the new contact. After a successful contact process, both the parties hold the same token linked to the identifier of the other. When two contacts interact with each other, the token is passed as an argument of the remote call. The receiving party looks up the token in the list of contacts and verifies the clearance of the required information.

3.3.3 Functions of a SPF Provider

SPF Provider is the interface which allows users to control the framework and its interaction with the external world. In particular, by means of SPF Provider, users can review and modify the social information available in the framework, and control how this data is made available to applications installed on the same device and remote instances of SPF. Figure 3.4 shows the interactions between the user and SPF by means of SPF provider and local applications. In the following, we detail the main features available to users in SPF Provider.

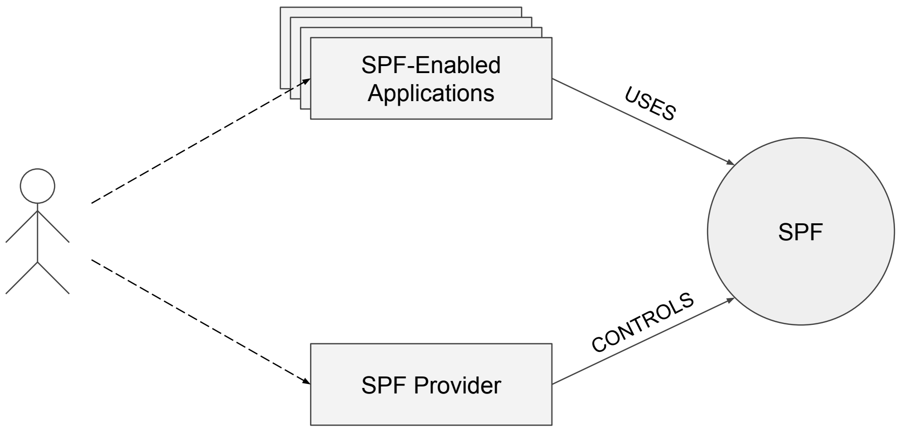


Figure 3.4: Interaction between the user, applications and SPF

Differently from other mobile social frameworks, SPF Provider allows users to directly control the information stored in their social profile, discussed in section 3.2.1. In fact, since local applications may be allowed to modify field values, the user should be able to preview what is shared with others about him, so that any rogue piece of data can be removed.

In order to limit the control of local applications on the user profile, each application is assigned a profile persona during the approval process previously described. This persona is used to determine the value of a profile field that can be read and written by the application. This also enables profile differentiation among applications of different domains. By means of SPF provider, the user can control the list of available profile personas, preview the field values contained in each of them and modify these values.

The user can also restrict the access to his profile information from remote instances by means of groups. For each profile field, the user can select which groups are allowed to obtain the field value: when a read request for a specific field is received, the framework returns the field value only if the sender belongs to a group which has visibility on the field. Groups can be managed by means of SPF Provider: the user can create new groups or remove existing ones, and assign each contact to one or more groups.

Personas are employed in remote access control as well: in fact, each read request, contains the identifier of the application that generated it. In this way, if the remote application is installed on the local device, the field value to return is obtained from the persona assigned to the local instance of the application. If the application is not installed, a default persona will be used.

The same level of control is available also on local applications, as the user can review the list of applications previously allowed and revoke access if needed; once the permission has been revoked, the application is prevented from accessing the framework functions until the access is granted again. For each allowed application, the user can obtain the list of permissions required by the application, together with its registered services.

The framework also allows the user to control the routing of SPF Activities, discussed in section 3.2.5. In particular, SPF provider displays the list of activity verbs supported by local applications. For each verb, the user can access the list of services that can consume it, and select which one should be the default target for activities of that verb. For example, a user that has installed several chat applications may decide which one should receive incoming messages.

Chapter 4

System Architecture

Software and cathedrals are much the same
first we build them, then we pray.

Anonymous

This chapter presents the architecture of SPF, a framework that enables the creation of *social smart spaces*, according to the requirements described in chapter 3.

Section 4.1 provides a high level overview of the architecture of SPF Provider, introducing the Core Layer, which implements the framework functions, and the interfaces that enable the communication with external entities.

Section 4.2 describes the internal components that compose the Core Layer, and how they cooperate in order to enable the framework functions.

Section 4.3 describes the Local Application Interface, that enables the communication between the SPF Core Layer and SPF-enabled applications mediated by the SPF Library.

Section 4.4 presents the Middleware Interface, which abstracts the underlying networking technology used to communicate with other SPF instances. Two implementations are then described, the first based on the AllJoyn middleware, the second on WiFi Direct.

4.1 Overview

A SPF-enabled smart space is constituted by a series of devices running SPF Provider and applications that offer proximity services. Figure 4.1 shows the runtime view of a smart space composed by two devices. Each device runs an instance of SPF Provider, and two external applications that leverage on the framework to offer proximity services.

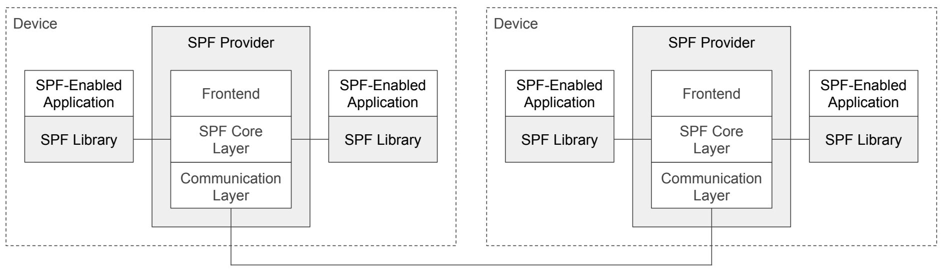


Figure 4.1: Runtime View

The interaction between different applications, either installed on the same device or on remote ones, is mediated by SPF, which manages the network communication. The SPF Library offers a set of tools that enables the interaction with the Provider. The aim of these tools is easing the development of social proximity applications by providing reusable components and hiding the communication details. Appendix A presents the API of the library.

SPF Provider is the application that offers the framework functions. It is structured on three layers. From bottom to top, the first is the Communication Layer, which enables the communication with remote SPF instances. This function is implemented by communication primitives that are made available to the upper layer.

The second is the Core Layer, which contains the business logic that realizes the functions described in the previous chapter. These functions leverage on the capabilities offered by the Communication Layer for the interaction with remote instances.

The third is the Frontend Layer, which allows the user to access the framework functions contained in the Core layer by means of a graphical user interface. In particular, the capabilities offered by this layer have been discussed in section 3.3.3.

The Core Layer also exposes the entry point for the communication with local applications. In order to interact with the Core layer, applications must integrate the SPF Library. This Library is a collection of software tools that provide access to the SPF functions available to apps, described in section 3.2.

A more detailed representation of the layered architecture of SPF is provided in figure 4.2. The central component of the architecture is the Core Layer which comprises a series of internal components. The interaction with the functions of the Core Layer happens by means of software interfaces that can be accessed by other components.

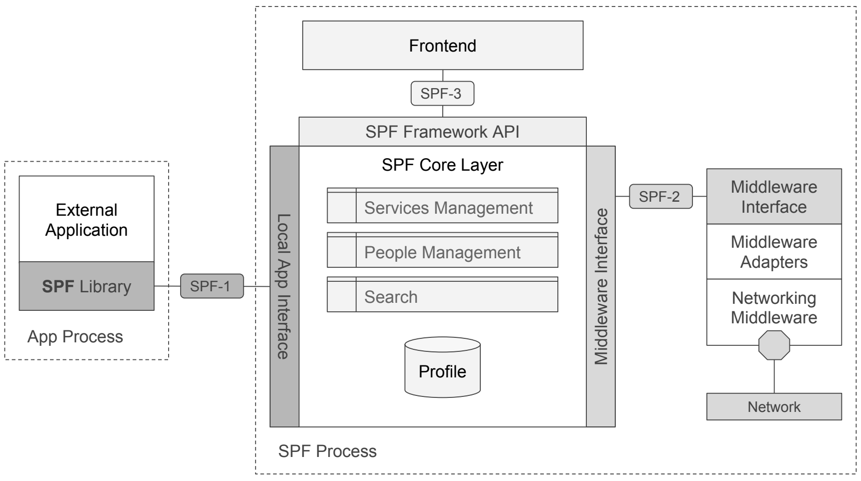


Figure 4.2: Architecture View

The frontend layer interacts with the core layer through an application programming interface named **SPF-3**, which is specified in Appendix B. The interaction with the Communication Layer (depicted on the right) and local applications (on the left) are mediated by software interfaces described in the following subsections.

4.1.1 Local Application Interface

The interactions between the Core Layer and local applications happen by means of the **SPF-1** interface, shown in figure 4.2. This is a bidirectional interface that allows both local applications and SPF to interact with the other party. Given that each application is run in its own process, this interface relies on the Inter-Process Communication (IPC) facilities offered by the underlying operating system.

In the case of current implementation, the Android operating system provides a set of tools for inter-process communication between applications. One of the main tools is Android `Services`, general-purpose software components that are used to perform operations in the background without the user intervention. The life-cycle of Android Services is managed automatically by the operating system, independently from other application components. In particular, the operating system can dispatch a message to a service even if the host application is not running at the moment. The target service, in fact, is activated when the request arrives, and deactivated as soon as it is no more needed. Each `Service` is identified by a `ComponentName`, composed by the package identifier of the application and its fully qualified class name.

One of the mechanisms to connect to a service is *binding*. In order to bind to a service, a component needs to know the service component name. When an application component binds to a service, it obtains a reference to an object called `Binder` that enables the inter-process invocation of methods declared by the service. Therefore, the applications involved in the communication must first agree on a shared programming interface, defined by means of the Android Interface Definition Language. Each AIDL interface defines the list of methods exposed by a service that can be executed by other components. Once the interface has been defined, the communication infrastructure that enables the remote invocation is automatically generated.

Another way to implement IPC between applications is using `Intents`, abstract descriptions of an operation to be performed. Intents can be dispatched to other application components or to the operating system itself, according to a publish subscribe model.

The Android implementation of SPF leverages on these capabilities to implement the interaction with local applications. Section 4.3 provides a detailed description of the **SPF-1** interface and its implementation.

4.1.2 Middleware Interface

The SPF Core Layer relies on an abstract Communication Layer that makes it independent from the actual networking technology used for the layer implementation. The interface exposed by this layer, named `SPF-2`, is shown in figure 4.2. This interface, as the one previously described, is bi-directional: on one side, it allows the Core Layer to control the middleware functions. On the other side, the interface enables the dispatch of events from the middleware to the upper layer. A detailed description of the middleware interface, together with two example implementations, are described in section 4.4.

The abstraction provided by the `SPF-2` interface enables a middleware-agnostic implementation of SPF, which benefits the portability of the framework. In fact, SPF can be ported to run on top of existing communication technologies by implementing suitable adapters: no changes to the infrastructure of the Core Layer are required. This means that the framework can be easily adapted to contexts in which a particular communication technology is required. For example, an implementation on WiFi is tailored to infrastructure-based smart spaces, while one based on WiFi Direct or Bluetooth is tailored to pure P2P scenarios.

Since networking technologies evolve rapidly, this approach also aims at easing the integration of novel technologies that will be available in the future, minimizing the impact on the rest of the infrastructure. Once SPF is ported to these technologies, it will leverage their improved performance and capabilities.

4.2 Internal components

The SPF Core Layer comprises a series of components that implement the framework capabilities. These software modules can be grouped according to their functionality as follows:

Security : components that implement the access control to the resource of the framework.

Services : components that coordinate the registration and execution of SPF Services.

Profile : components that store the profile information and provide read and write capabilities.

Search : components that implement the discovery capabilities and coordinate search processes.

Advertising : components that enable the advertising of a subset of the user profile and the definition of triggers.

Activities : components that implement the dispatching of activities.

4.2.1 Security

SPF implements access control mechanisms that gives the user full control on how local applications and remote instances interact with the framework. These mechanisms are implemented within the communication interfaces that provide access to the SPF Core Layer. In particular, the Local Application Interface (section 4.1.1) defines the access control for applications, while the Middleware Interface (section 4.1.2) the one for remote instances. These two mechanisms are coordinated by the `Security Monitor`, a component available in the Core Layer.

The aim of application access control is to make the user aware of new applications that interact with the framework, so that he can directly authorize them. Moreover, for a specific application, the user can control which SPF functions can be accessed and which subset of the user resources can be accessed. This is obtained by means of a token based identification system: upon each interaction with the framework, an application needs to provide a token that enables SPF to identify the source of the request. The identification token is unique for each application, and is granted after the application undergoes a registration process, shown in figure 4.3.

The registration process is transparent and asynchronous, and is triggered automatically when an application performs his first interaction with the framework, in order to obtain the identification token. In the first step of the process, the library retrieves a series of details about the application and includes them in an `AppDescriptor`. These details include the application identifier, unique among all applications on the device, and the set of permissions required by the application, as discussed in section 3.3.1. Once the required information is available, the library dispatches a registration request to the Core Layer, providing the descriptor.

When the Core Layer receives a registration request, it first validates the information contained in the descriptor, checking if an application with the provided identifier is actually installed on the device. Then, it retrieves the name and the icon of the application that matches the identifier: this information is displayed in a dialog presented to the user, which also lists the permission required by the application. By means of this dialog, the user an choose whether to grant or not access

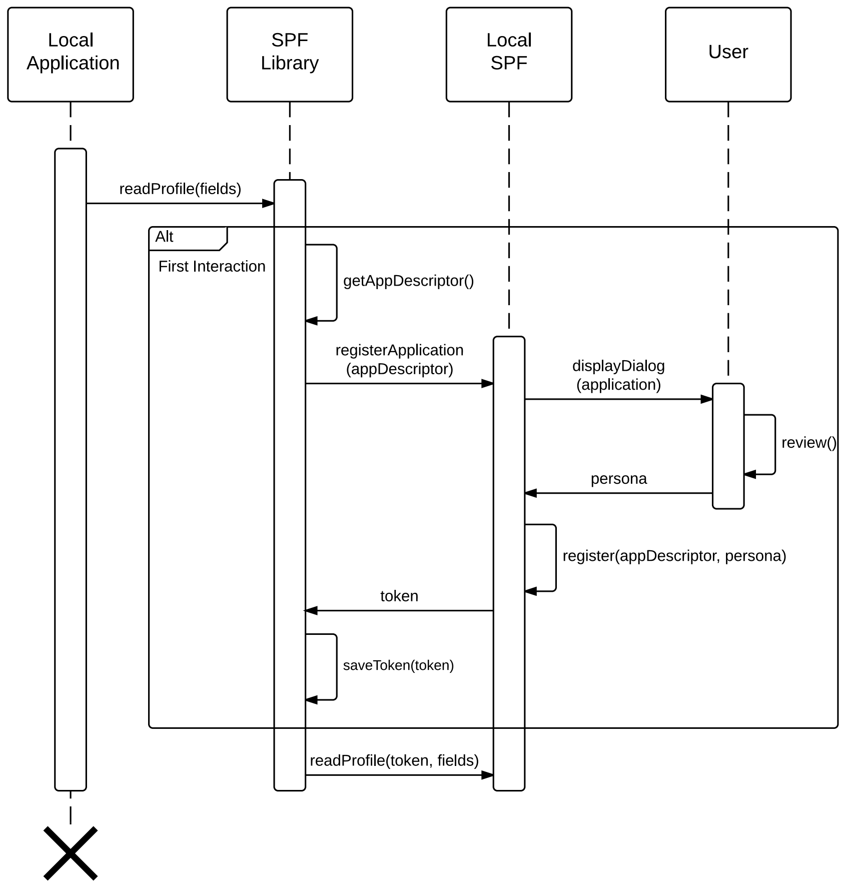


Figure 4.3: Authorization of a local application

to the framework. The dialog also allows the user to select the profile persona that the new application will be able to access.

If the registration request is approved, the details of the newly approved application are stored in the Security Monitor, which provides an `ApplicationRegistry` for this purpose. For each application, the registry stores the identifier, name, profile persona, and the list of permissions that have been granted. Then, the Security Monitor generates an unique identification token, stored together with the other application details. Once the registration is completed, the token is returned to the library to be used for future interactions. Upon reception, the library stores the identification token, and carries on the original interaction that triggered the process. The next time an interaction is required, the library will retrieve the stored token and use it directly, without triggering the registration process.

The user may also deny the access to an application: in this case, the application is not registered, and an error notification is given back instead of the token. When the notification is received, the library notifies the application that the authorization was denied, and the original interaction fails.

All the methods exposed by the Local Application Interface, except for the one to start the registration process, require an identification token. When the Core Layer receives a request, it looks up the application details from the Security Monitor: if a record for the given token exists, the Core Layer checks whether the application was granted the permission for the requested interaction. If this is the case, the interaction is performed, and the result returned, otherwise the interaction is blocked, and an error is returned.

The other access control mechanism is introduced as a mean to restrict the access of remote instances to local resources. As discussed in the previous chapter, this mechanisms is built upon the concept of *Groups*, collections of remote instances that can be granted access to specific resources. The identification of remote instances is realized by means of *Contact Tokens*, in a similar way to the application access control: two SPF instances are “in contact” when they share the same token. The token is exchange with a procedure that requires the approval of both users.

The remote access control in manages by the *Person Registry*, a component exposed by the Security Monitor. When the user sends a contact request to another person, The Security Monitor generates a unique token and stores it in the Person Registry, together with the basic profile details of the remote instance (display name, identifier, profile picture and the groups to which the instance is assigned). Then, the token and the same personal details about the local user, obtained from the profile, are packaged into a Contact Request, and sent to the other instance through the middleware.

Incoming Contact Request are managed by the Security Monitor as well. When a new request arrives, the profile details of the sender are read from the request and displayed to the local user, which can accept or deny the request. In case the request is accepted, the token and the profile details are stored in the Person Registry, otherwise the request is discarded. The process does not include a response, as it is designed to be as short as possible.

Once both the instances share the same Contact Token, it can be used to access remote resources with privileged access. In case of profile access, when the local instance dispatches a remote read request to a remote instance, it includes the shared token. When the request is received, the remote instance uses the token to lookup the groups to which the remote instance is assigned. Once this piece of data is available, the information is read from the profile, as described in section 4.2.3. It is possible to dispatch read requests to an instance that is not in the contact list. In this case, no token is included in the request, and only the public profile fields will be accessible.

4.2.2 Services

The SPF Core Layer provides the infrastructure to enable the registration and execution of SPF Services. As described in section 3.2.2, SPF Services are software components defined by local applications that are published in proximity by SPF, so that other applications can invoke them. The component of the Core Layer that manages and coordinates the registration and execution of services is called `ServiceRegistry`, and its architecture is shown in figure 4.4.

The functions of the `ServiceRegistry` include the registration and deregistration of services, the invocation of registered SPF Service and the dispatch of incoming SPF Activities (discussed in section 4.2.6). These functions are implemented by means of the following three components:

`ServiceTable` stores the information about registered services. This is necessary to retrieve the information required to invoke a service when a remote request arrives.

`ActivityRouteTable` stores the information about services that can consume SPF Activities, and the default service for each activity verb.

`AppCommunicationAgent` manages the connections to the local applications used to dispatch service invocations and activities. The `AppServiceProxy` class abstracts the connections to the remote components.

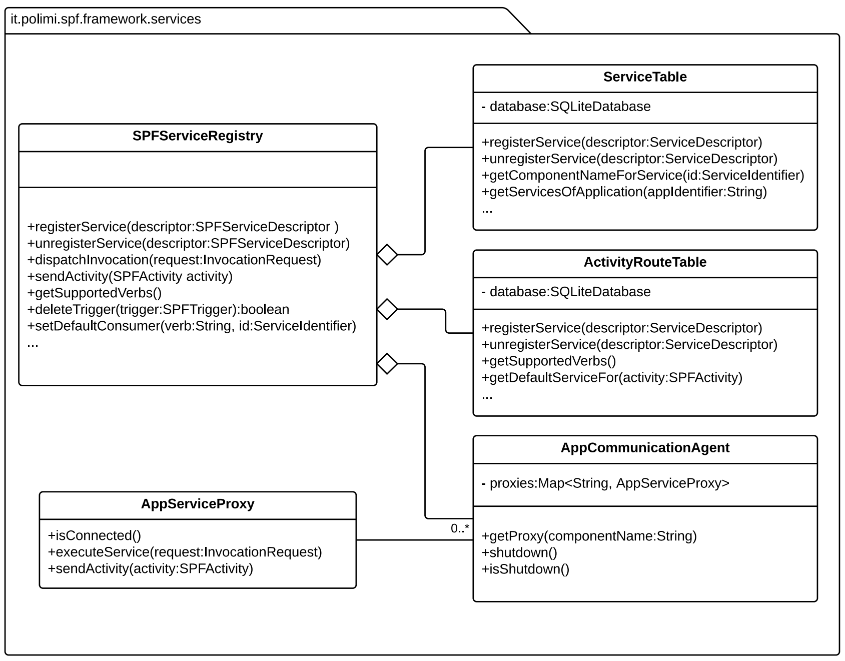

it.polimi.spf.framework.services
SPFServiceRegistry
+registerService(descriptor:SPFServiceDescriptor)
+unregisterService(descriptor:SPFServiceDescriptor)
+dispatchInvocation(request:InvocationRequest)
+sendActivity(SPFActivity activity)
+getSupportedVerbs()
+deleteTrigger(trigger:SPFTrigger):boolean
+setDefaultConsumer(verb:String, id:ServiceIdentifier)
...
ServiceTable
- database:SQLiteDatabase
+registerService(descriptor:ServiceDescriptor)
+unregisterService(descriptor:ServiceDescriptor)
+getComponentNameForService(id:ServiceIdentifier)
+getServicesOfApplication(appIdentifier:String)
...
ActivityRouteTable
- database:SQLiteDatabase
+registerService(descriptor:ServiceDescriptor)
+unregisterService(descriptor:ServiceDescriptor)
+getSupportedVerbs()
+getDefaultServiceFor(activity:SPFActivity)
...
AppCommunicationAgent
- proxies:Map<String, AppServiceProxy>
+getProxy(componentName:String)
+shutdown()
+isShutdown()
AppServiceProxy
+isConnected()
+executeService(request:InvocationRequest)
+sendActivity(activity:SPFActivity)
0..*

Figure 4.4: Architecture of the SPF Service Registry

An application can define a SPF Service leveraging on the tools offered by the SPF library. To define a service applications are required to provide a programming interface that defines the methods available for execution. This interface should also be annotated to provide required meta-data about the service, including its name. Remote applications that want to invoke methods of a service must know its interface and provide it to SPF.

Once the interface has been defined, the application must provide an implementation leveraging on `SPFServiceEndpoint`, a specialized Android Service that realizes the application side of the remote invocation mechanism. To define a service implementation, the application must declare a concrete subclass of `SPFServiceEndpoint` that implements the desired service interface. When the implementation is instantiated by the operating system, the business logic in the endpoint detects the implemented service interface and creates an index, containing the references to all the concrete methods. During the execution of an invocation request, the endpoint retrieves the method from the index and, using the Java Reflection API, invokes the required method. To dispatch incoming invocation requests, SPF Provider binds to the concrete endpoint of the SPF Service. After the bound is complete, the provider obtains a `Binder` that enables the inter-process dispatching of the invocation request. Like standard Android Services, each subclass of `SPFServiceEndpoint` is identified by a Component Name, required during the binding process.

After the implementation and the interface have been defined, the application can register the service in SPF by means of a registration process. This process starts when the local application requires to register a service providing its interface and implementation. Before interacting with the Core Layer, the library checks the interface to verify that both the meta-data and declared methods are valid. If the validation is successful, the registration request is dispatched to the `ServiceRegistry`, which stores its information in the `ServiceTable`. In particular, the table stores the name of the service and the identifier of the app that registered it: this two pieces of data are used to identify the service. The table also stores the Component Name of the Service Endpoint that contains the implementation. Once the service information is stored in the registry, the service is ready to be executed by external applications.

Applications can execute remote services leveraging on the functions of the SPF Library. The invocation of a service is enabled by an `InvocationStub`, a proxy object that implements the same interface as the remote service. This object permits the execution of remote methods as if they were local, according to a remote method invocation model. In this way, all the communication details are hidden, including the inter-process communication with the core layer and the transmission of the request to the remote instance. To obtain an invocation stub for a remote service, an application must select the instance that is publishing the service, and provide the interface. Figure 4.5 shows how an invocation request is dispatched to the corresponding `SPFServiceEndpoint`.

When the stub receives a method call, it creates an `InvocationRequest` containing the information needed for the remote execution. This information includes the identifier of the application that published the service, and the identifier of the service. If the method signature defines input parameters, their values, supplied by the invok-

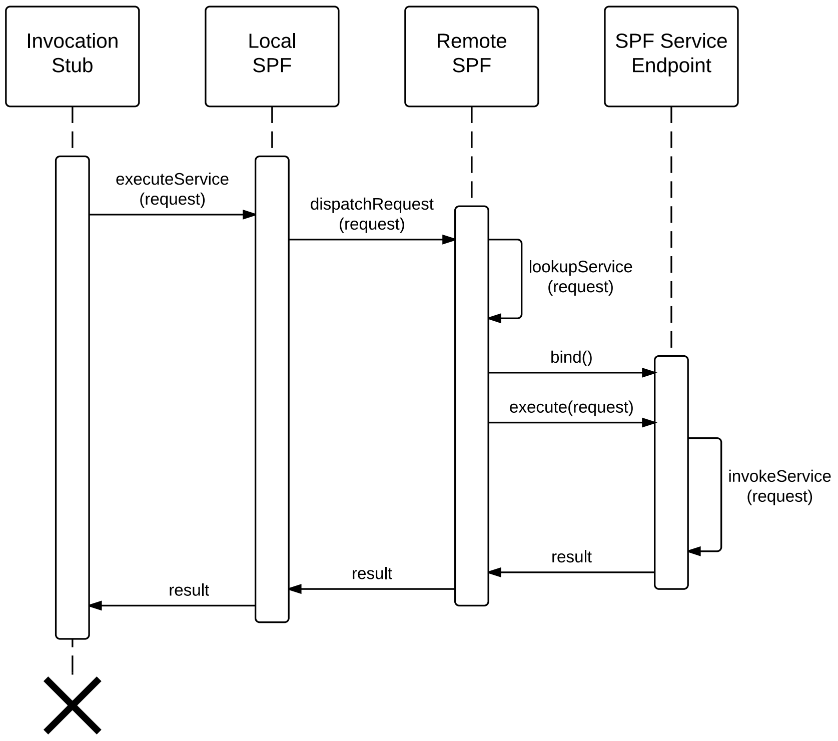


Figure 4.5: Execution of a SPF Service

ing application, are serialized and added to the request. Once ready, the request is delivered to the remote SPF instance. This delivery consists of two steps: first, the request is dispatched to the local SPF instance by means of the local application interface. Then, the request is delivered to the remote instance through the middleware.

Upon reception, the SPF Core Layer dispatches incoming request to the `ServiceRegistry`. This component first looks up the requested service in the `ServiceTable`, in order to retrieve the component name of the Service Endpoint. Once the component name is retrieved, the registry obtains a proxy to communicate with the local application from the `AppCommunicationAgent`. This component is in charge of binding to the service endpoints of local applications, obtaining the Binder needed to dispatch invocations. These binders are wrapped by instances of the `AppServiceProxy` class, which enables a finer control on the life-cycle of the binder. In particular, since the binding process is asynchronous, the proxy is used as a serialization point where the execution process is blocked until the `Binder` object is ready. Active proxies are maintained in a cache for a predefined time interval. If the proxy is requested again before the timeout, it is retrieved from the cache and the timeout is reset. Once the timeout expires, the connection is closed and the proxy removed from the cache. Once the `Binder` is obtained, the registry dispatches the request to the endpoint.

Once the request is received, the service endpoint looks up the method to invoke from the method index using the name contained in the request. If the method requires parameters, their values are deserialized from the request payload, then the method is invoked. Once the invocation is concluded, the endpoint creates an `InvocationResponse` to be sent back to the application that performed the invocation. If the method supports a return value, the result is serialized and included in the payload of the response. Then, the response is sent back to the sender of the request by means of the same two-step delivery mechanism used to dispatch the request. The invocation response package is also used to signal errors that may occur in any step of the invocation process. Errors may be generated either locally (e.g, the remote instance is not found in proximity) or remotely (the service does not exist, or an exception is thrown during the execution of the method).

4.2.3 Profile

The SPF Profile provides a persistent social profile accessible to local applications and remote instances according to the visibility preference set by the user. To restrict the access by remote instances, the user can define group of contacts and allow only a subset of the groups to access a profile field. The user can also diversify the information provided to local applications by means of profile personas.

The user profile is managed by a component of the Core Layer called `Profile Manager`, whose three main functions are storing the profile data, providing available profile personas and maintaining the list of profile field visibility.

The profile information is divided into profile fields, and each field value is stored in a double key data structure, where the a key is the field identifier, and the other is the persona. In this way, the data structure can hold the profile information of all personas. The list of available personas, on the other hand, is kept in a separate table. A default persona is always available, but users can create new ones by means of the front-end.

The Profile Manager also stores the visibility of each profile field to groups of contacts. When a remote instance requires to access the values of a series of profile fields, the Profile Manager obtains the list of fields visible to that instances, removes from the requested list of fields those not visible, then return the visible values to the remote instance.

Remote read requests also contain the identifier of the application that sent the request. In this way, if the application is installed on the local device, the field values are obtained from its associated profile persona, otherwise the default one is used.

4.2.4 Search

The Search functionality provided by SPF allows applications to discover nearby people according to queries. The visibility of instances depends on the middleware used to communicate: for example, if a middleware based on standard Wi-Fi is used, the search will discover instances connected to the same Wi-Fi network.

To start a discovery process, an application must provide a query object, a configuration and a callback. The query contains a set of parameters that should be matched by remote instances that are discovered by the process, as described in section 3.2.3. The search configuration is used to control how search requests are sent through the middleware. Finally, the callback contains the business logic defined by the application to react to search events.

The instance discovery process is implemented in a distributed way that delegates the evaluation of search queries to remote instances, as shown in figure 4.6. The communication between instances happens through search signals sent through the middleware. The instance in discovery sends multicast search signals to all reachable instances. These signals contain the serialized query and details about the sender. When an instance receives the search signal, the query is deserialized and evaluated against the profile. If the query is matched, a search result is sent back to the instance in discovery.

Ongoing discovery process are managed by the `SPFSearchManager`, a component contained in the SPF Core Layer. The search Manager keeps track of all active searches and of results dispatched to applications. In this way, when the middleware notifies that an instance is no more reachable, the Search Manager can notify interested applications

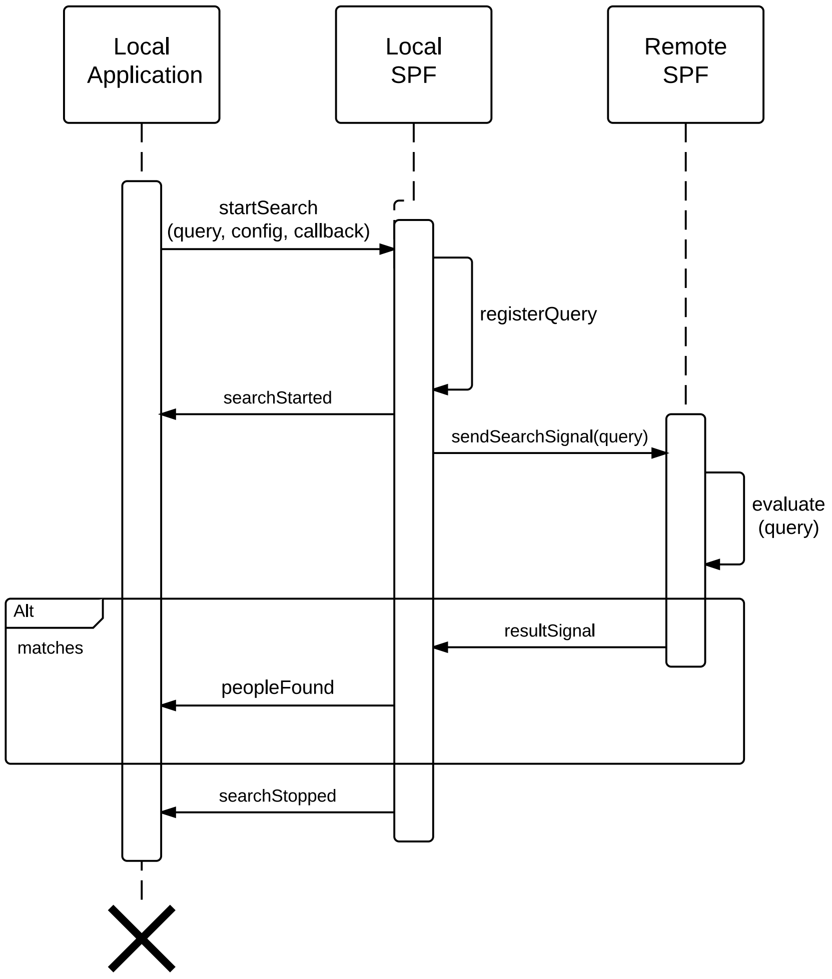


Figure 4.6: Execution of a search process

of the event. The dispatched results list is also used to notify each result once to applications.

When an application starts a search, the Search Manager saves the request in the list of active searches and generates a query ID, that is used to associate search signals with the corresponding results. Then, it notifies the application that the search has started, then begins the dispatching of search signals. The number of signals and the time interval between two subsequent ones are defined in the search configuration provided by the application. Once all signals have been sent, the search is removed from the list of active searches, and the list of results delivered for the concluded search is also cleared.

The Search Manager is also in charge of handling incoming search signals from remote instances. Once the query is de-serialized from the signal payload, it is passed to the *Search Responder*, a component that verifies whether the local instance matches the query or not. In case of positive match, a result signal is sent back to the instance in discovery. The search result signal includes the basic profile details of the local instance (display name and identifier) obtained from the Profile Manager.

The search process is asynchronous, and doesn't block the application that started it: the notifications of search events are dispatched by the Search Manager using the callback provided by the application. Since the Search Manager and the local application are hosted in different processes, the dispatching of events requires inter-process calls. However, the infrastructure to perform these calls is provided transparently by the framework.

4.2.5 Advertising

SPFAdvertising allows users to advertise their own profile. This functions can be configured from the front-end of the SPF Provider, which offers a dedicated interface for the selection of profile contents that the user wants to advertise. The list of installed applications can also be included in the advertisement. Once enabled, the advertising is tied to the life-cycle of the proximity middleware: when SPF is about to setup the connections, it checks if the SPFAdvertising is active and, if so, it requests the middleware to broadcast the advertisement. The proximity middleware is in charge of broadcasting the information as long as the advertising is stopped or network resources are released.

`SPFTrigger` is the abstraction that represents the rule that can be used to react upon the receipt of an advertised profile. A trigger is made up by two components: a query, as the one specified for the search, and an action. Figure 4.7 show the structure of a trigger. There exist two types of action that can be executed either on the local device or on the remote ones:

- `SPFActionIntent`: broadcasts an Android Intent that can be received from the application that registered the trigger. The Intent action name is specified by the application.
- `SPFActionSendMessage`: sends a message to the remote instance

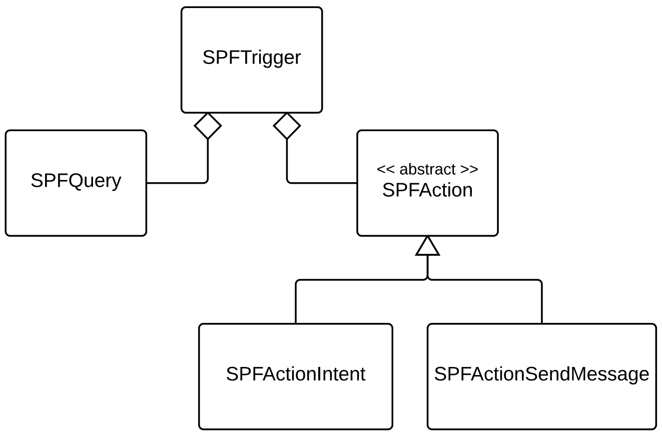


Figure 4.7: Structure of SPFTrigger

that has activated the trigger. SPF Provider will show a textual message in an Android notification.

Each action inherits from `SPFAction` which is one of the property of `SPFTrigger`; other actions can be added by creating a new subclass of `SPFAction` and by providing an implementation of the action itself.

`SPFNotificationManager` is the component that keeps and coordinates the objects that implement SPFNotification service. Figure 4.8 shows a class diagram of these components.

`SPFTriggerTable` is the class that offers the store procedures of the database that holds triggers information.

`SPFTriggerEngine` is the component that executes the business logic about triggers: compares the queries with the received advertisements and in case of positive match signals the event to the `SPFActionPerformer` interface.

`SPFActionCache` stores information about performed action; this is necessary to avoid repeated action on a same target. This property can be configured by setting the sleep period of a `SPFTrigger`.

`SPFActionPerformer` is the interface that defines the method to perform `SPFAction`, its actual implementation (not shown in figure) is provided by SPFActionPerformerImpl class and held by SPFNotificationManager.

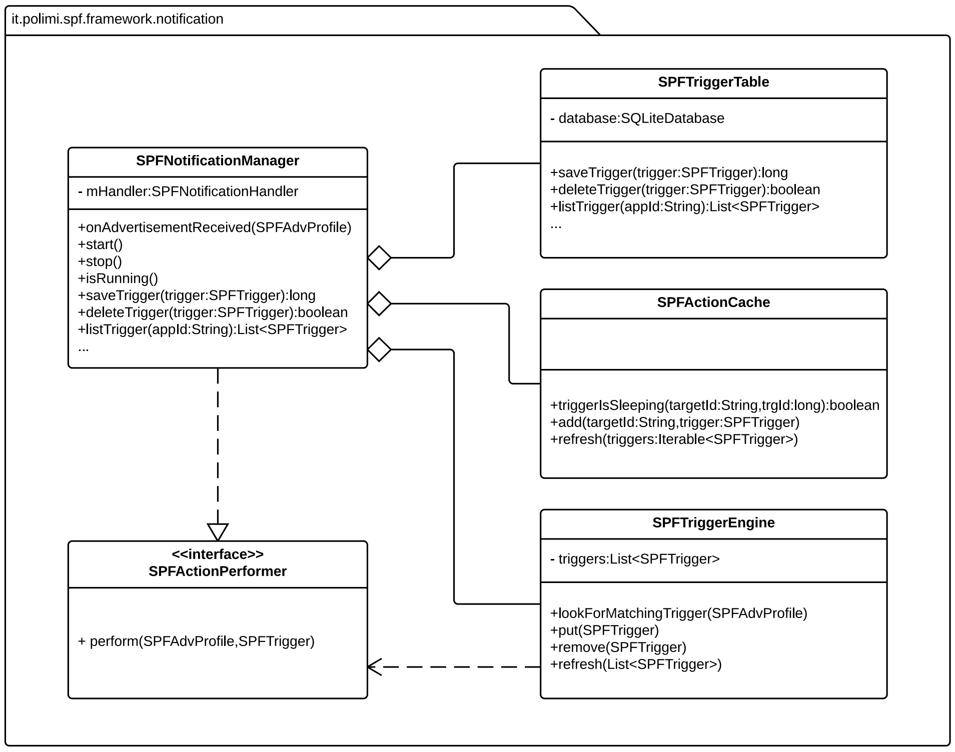

it.polimi.spf.framework.notification
SPFNotificationManager
- mHandler:SPFNotificationHandler
+onAdvertisementReceived(SPFAdvProfile)
+start()
+stop()
+isRunning()
+saveTrigger(trigger:SPFTrigger):long
+deleteTrigger(trigger:SPFTrigger):boolean
+listTrigger(appId:String):List<SPFTrigger>
...
SPFTriggerTable
- database:SQLiteDatabase
+saveTrigger(trigger:SPFTrigger):long
+deleteTrigger(trigger:SPFTrigger):boolean
+listTrigger(appId:String):List<SPFTrigger>
...
SPFActionCache
+triggerIsSleeping(targetId:String,trgId:long):boolean
+add(targetId:String,trigger:SPFTrigger)
+refresh(triggers:Iterable<SPFTrigger>)
SPFTriggerEngine
- triggers:List<SPFTrigger>
+lookForMatchingTrigger(SPFAdvProfile)
+put(SPFTrigger)
+remove(SPFTrigger)
+refresh(List<SPFTrigger>)
<<interface>>
SPFActionPerformer
+ perform(SPFAdvProfile,SPFTrigger)

Figure 4.8: Class diagram of the Notification package

Concurrency and synchronization issues are resolved by using a single thread, managed by an Android `Handler`, to access the components that holds the triggers logic. By means of this Handler thread all the processing is parallelized with a queue of events that decouples the components from the external ones.

`SPFNotificationManager` lifecycle is tied to the one of the proximity middleware: `SPFActionCache`, `SPFTriggerEngine` and the internal handler thread are initialized and destroyed with start() and stop() methods, accordingly to the state of the network resources. Before the call to start() and after the one to stop(), only `SPFTriggerTable` is in a legal state. This allows us to save resources when there is no need of instantiating the other components, since the receipt of an advertisement can happen only when the proximity middleware is in an active state.

4.2.6 Activities

As discussed in section 3.2.5, SPF supports *Activities*, inter-operable data structure containing standardized information that can be used by applications to communicate well-known social actions performed by users. Activities are characterized by a verb that identifies the type of social action described and is used to determine to which application the activity will be dispatched.

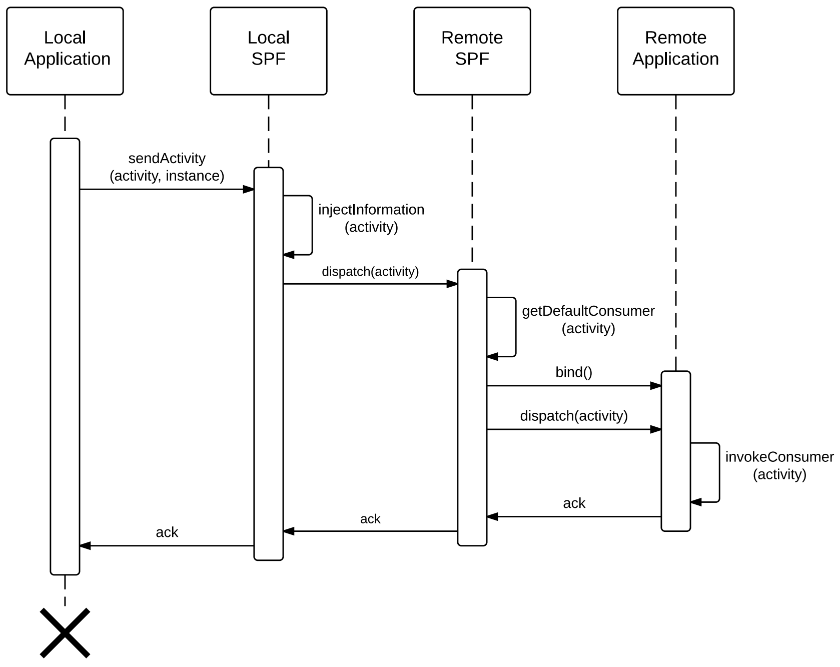

Local Application
Local SPF
Remote SPF
Remote Application
sendActivity (activity, instance)
injectInformation (activity)
dispatch(activity)
getDefaultConsumer (activity)
bind()
dispatch(activity)
invokeConsumer (activity)
ack
ack
ack

Figure 4.9: Dispatch of an activity

The dispatching of SPF Activities is implemented within the SPF Service infrastructure, shown in figure 4.4. A SPF Service, in fact, can be declared as a *consumer* of a series of activity verbs by means of specific annotations. For each verb that can be consumed, a service must declare a method to handle incoming activities dispatched by SPF. This method is recognized by means of an annotation that states which verb is supported by the method. The implementation, on the other hand, is provided within the Service Endpoint of the service, as with normal methods.

The details about services that can consume activities is stored in the `ActivityRouteTable`. When a service is registered, the registry checks whether it is declared as an activity consumer. In this case, the service is added in the `ActivityRouteTable`, with an entry for each supported verb. The Activity Route Table also lists the user-defined default service for each verb; the user can access this configuration by means of the front-end of SPF Provider.

The process to dispatch an activity, shown in figure 4.9, is similar to the invocation of a remote service. First, an application needs to create a new activity instance specifying the verb; further information can be added, according to the type of activity. Once the activity has been created, the application can dispatch it to a specific remote instance. Before the activity is sent, the Core Layer automatically injects pieces of information, including the display name and identifier of the sender and the creation time. Then, the activity is serialized and sent to the remote instance through the middleware.

Incoming activities are handled by the `ServiceRegistry` as well: upon reception, the registry is deserializes the activity and looks up its verb into the `ActivityRouteTable` to determine the default service. Then, the registry obtains the `ComponentName` for the default service from the Service Table, and obtains a proxy to the `ServiceEndpoint` from the `AppCommunicationAgent`, similarly to the dispatching of an invocation request. Once the proxy is available, the activity is dispatched to the endpoint, which looks up the method designed to handle the activity, and invokes it.

Once the invocation is concluded, an acknowledgment message is sent back to the source of the activity using an Invocation Response. This response is also used to signal any error occurred during the dispatch of the activity.

4.3 Local Application Interface

As described in section 4.1.1, the communication between the Core Layer and local applications happens by means of the Inter Process Communication capabilities offered by the operating system. Android provides these capabilities with Services, application components that provide software interfaces to external applications to enable intercommunication.

In order to allow local applications to interact with the Core Layer, SPF defines an Android service named `SPFService`, which exposes multiple interfaces allowing the interactions previously described. These interfaces are:

`SPFSecurityService` : Provides the entry point for the registration of external applications.

`LocalProfileService` : Provides access to the Profile API in order to read and write field values on the local profile.

`LocalServiceManager` : Provides access to the Services API, allowing to register and deregister services, and execute those of local applications.

`SPFNotificationService` : Provides access to the Notification API to define triggers.

`SPFProximityService` : Provides the capabilities to interact with remote applications, including the search functions, execution of remote services and read access to remote profiles.

Local applications can access these functions even if the SPF Provider application is not running. In fact, once a local application requires to bind to the `SPFService`, the operative system automatically bootstraps the Provider's application context and activates the functions of the service. `SPFService` also manages the network resources of the framework by instantiating them when needed.

The SPF Library provides a series of *components*, shown in figure 4.10, that facilitate the interaction with the functions provided by the `SPFService`. These components wrap the stubs obtained from the operating system to communicate with the interfaces of the `SPFService`, also hiding the binding process needed to know the underlying protocols of interaction with SPF.

All the components provided by the SPF Library share a common superclass named `SPFComponent`. This class implements the process of binding to the SPF Service, including the application registration, as described in section 4.2.1. Once the registration process is concluded, `SPFComponent` stores the access token in the `AccessTokenManager`, so that it can be retrieved during future interactions. In this way, components can provide a simpler interface to the `SPFService`, as there is no need for applications to directly pass the access token.

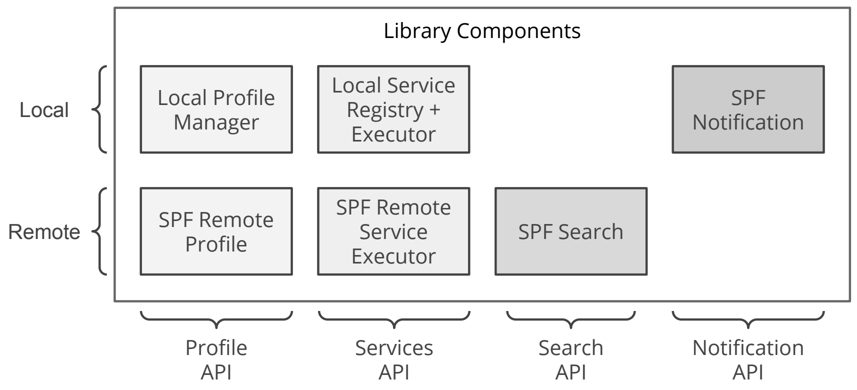


Figure 4.10: Components provided by the SPF Library

4.4 Middleware

As discussed in the previous sections, SPF defines a set of interfaces that allows the interaction with a generic networking middleware. In detail, there are three main interfaces that allows to control the middleware, receive external events and request, and interact with remote instances.

`InboundProximityInterface` defines the events that can be received by SPF. These includes search signals, service invocation requests, and profile remote readings.

`ProximityMiddleware` defines the commands that allows SPF to control the networking layer.

`SPFRemoteInstance` defines the remote calls that an instance of SPF can execute.

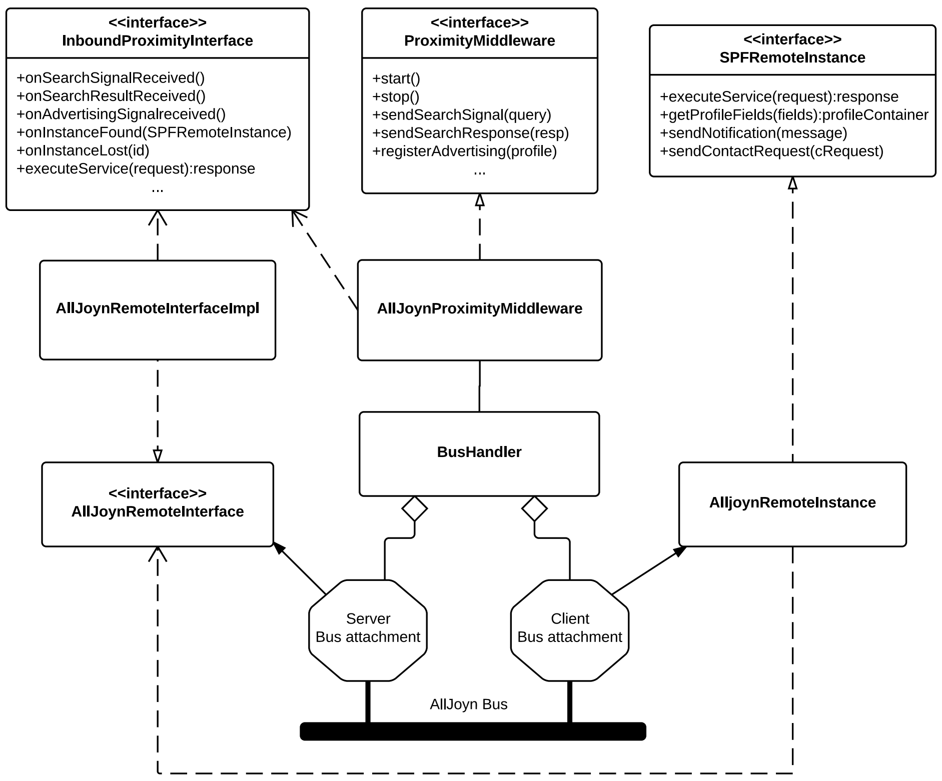
A networking middleware for SPF has to use the first interface to communicate with SPF and provide an implementations of the last two according the API discussed in appendix B. The following sections describe the architecture of two implementation based respectively on AllJoyn and Wi-Fi Direct. An introduction about these technologies can be found in chapter 2.

4.4.1 AllJoyn

AllJoyn is a distributed object-oriented middleware that provides a high-level abstraction based on the concept of distributed bus. Remote objects has to published over the bus by means of bus attachments, that are identified either by randomly assigned names or application defined identifiers, called well-known names.

SPF uses AllJoyn to expose its remote methods and to discover remote instances. Each remote instance advertise itself with a well-known name, composed by the application package name and the user identifier. Different instances of SPF can discover each other by listening to the distributed bus for an advertisement of such a well-known name. Once the instance is discovered, SPF may create a connection establishing an AllJoyn session. Then the communication follows standard remote method invocation mechanisms.

Figure 4.11 shows the main components that allows SPF to interact with AllJoyn, basically they are adapters over the API provided by the middleware.



<<interface>>
InboundProximityInterface
+onSearchSignalReceived()
+onSearchResultReceived()
+onAdvertisingSignalreceived()
+onInstanceFound(SPFRemoteInstance)
+onInstanceLost(id)
+executeService(request):response
...
<<interface>>
ProximityMiddleware
+start()
+stop()
+sendSearchSignal(query)
+sendSearchResponse(resp)
+registerAdvertising(profile)
...
<<interface>>
SPFRemoteInstance
+executeService(request):response
+getProfileFields(fields):profileContainer
+sendNotification(message)
+sendContactRequest(cRequest)
AllJoynRemoteInterfaceImpl
AllJoynProximityMiddleware
BusHandler
<<interface>>
AllJoynRemoteInterface
AlljoynRemoteInstance
Server
Bus attachment
Client
Bus attachment
AllJoyn Bus

Figure 4.11: Class diagram of the AllJoyn adapters

The class that directly interacts with AllJoyn is the `BusHandler`. This component handles the actions and the events that characterize the interaction between AllJoyn and SPF. The connection with the AllJoyn Bus is made through two distinct bus attachments; the first, called *client bus attachment* is used to discover and interact with remote instances; the second, called *server bus attachment*, publishes and advertises the remote object that represents the instance of SPF. These two components, combined together, enable the peer-to-peer interactions of SPF.

As shown in figure 4.11, the interfaces provided by SPF are used to communicate with the upper layer of the system. `AllJoynRemote-Instance` is an implementation of `SPFRemoteInstance`, it is a lazy initialized proxy that mediates the access to the AllJoyn actual proxy bus object[1]. Whenever AllJoyn detects the presence on the bus of a remote instance, as well as its disappearance, the `InboundProx-imityInterface` is notified with the methods `onInstanceFound` and `onInstanceLost` with which a reference to an `SPFRemoteInstance` is passed. This reference keeps all the information that can then be used by SPF to create a connection.

As mentioned before, `AllJoynRemoteInstance` holds a reference to a proxy bus object that provides an interface for the remote method invocation. This interface is called `AllJoynRemoteInterface` and with its implementation mediates the access to SPF through the `Inbound-ProximityInterface`.

Generic middleware events like advertisements and search signals, are not associated with a specific proxy bus object and thus they are handled internally by the `BusHandler`. Their implementation is based on AllJoyn signals, which are means to transmit information to all the reachable devices interested in the type of signal.

4.4.2 Wi-Fi Direct

To show how SPF can benefit from emerging device-to-device network technologies, we designed another implementation of the networking layer that substitutes AllJoyn with Wi-Fi Direct. As described in section 2.1.1, Wi-Fi Direct allows two or more devices to communicate over Wi-Fi without the need of a central access point (AP). The role of AP is assigned dynamically after a negotiation phase during which a P2P Group is created and a Group Owner is elected. The P2P Group

[1] *proxy bus object* is the Alljoyn terminology to indicate the client side stub of a remote object. See chapter 2 for details.

Owner is responsible for providing the software access point to which all the other member of the group may connect. The technology also supports service discovery at the network layer; this allows two devices to exchange information prior to the group creation process. Differently from AllJoyn, Wi-Fi Direct API do not provide high-level abstractions: the actual communication is still based on standard socket technology.

Unfortunately, we discovered that Wi-Fi Direct is not a good candidate for SPF. In particular, it made it hard to replicate the collaboration between discovery and search that was so successful in the AllJoyn implementation. Our first attempt was to implement the instance discovery events and broadcast signals with the service discovery facilities offered by the API of Wi-Fi Direct. In theory, this would have allowed us to create P2P Groups only when needed, thus providing a more scalable solution. In practice, some constraints of the Wi-Fi Direct implementation prevent us from actually following this approach.

The first constraint regards the peer discovery that, in general, is extremely slow, taking up to 60 seconds to provide an updated list of nearby devices; this is caused by an internal constant defined at the operative system level that limits the updates of the list of available peers. A second, and even more important constraint, is that the service discovery API are designed for static content and thus it is unfeasible to use them for implementing connection-less communication. This would imply to register the information to advertise each time it changes, but more important the other device has to restart periodically the discovery process, since there is no mechanism for notifying remote devices about the change. These consideration led us to take a different approach.

Our solution is based on the construction of an overlay network driven by the Wi-Fi Direct P2P Group. We build a star topology overlay whose root node is the P2P Group Owner, to which the other members are connected through a socket connection. This solution allows the system to easily detect the instances that joins the group and thus are reachable from SPF. The Group Owner takes care of dispatching instance discovery messages to all the members of the group; these events include the discovery as well the loss of a member within the group, in the same way as happens with the discovery of well-known names in AllJoyn. It allows us also to improve general messaging since the Group Owner can simply route the messages to their intended destinations.

Since the communication is socket based, we need to implement a middleware to offer a higher level of abstraction and communication primitives that suit the requirements of SPF.

The naming of instances resemble the mechanism of AllJoyn based on well-known names. An instance is identified by a name that contains a prefix, in common between all the instances of SPF, and an identifier which is unique for the specific instance. As previously discussed, names are dispatched by the group owner through instance discovery messages.

To allow communication between connected instances, the middleware offers three communication primitives that replace AllJoyn signals and the distributed object oriented model. In particular:

- `sendBroadcastMessage` is a primitive for sending broadcast messages; They are delivered to the group owner that dispatches the messages to all the member of the group. Query signals and instance discovery messages are handled with this primitive.
- `sendMessage` is a primitive for sending unicast messages.
- `sendRequest` is an RPC-friendly primitive that allows to send a message and wait for its response. This is used to implement the RMI mechanism that was offered by AllJoyn.

Figure 4.12 shows a class diagram representing the main components of the middleware based on Wi-Fi Direct.

Due to the roles within a group are assigned dynamically, the design takes into account this complexity by providing a hierarchy of components that hides the different behaviors of the roles. `GroupActor` is the abstract class that defines the interface in common to a group owners and standard group members. It holds the logic that allows the middleware to deliver messages to the application.

`GroupClientActor` and `GroupOwnerActor` inherit from the previous abstract class and provide a concrete implementation regarding the actual network connections. In particular `GroupClientActor` is the class that implements the role of a standard group member, as such its solely duty is to connect to the group owner and offer functions for sending and receiving messages through its socket connections. Instead the `GroupOwnerActor` class is more complex and adds an additional layer over the socket connection for handling the specific functions of a

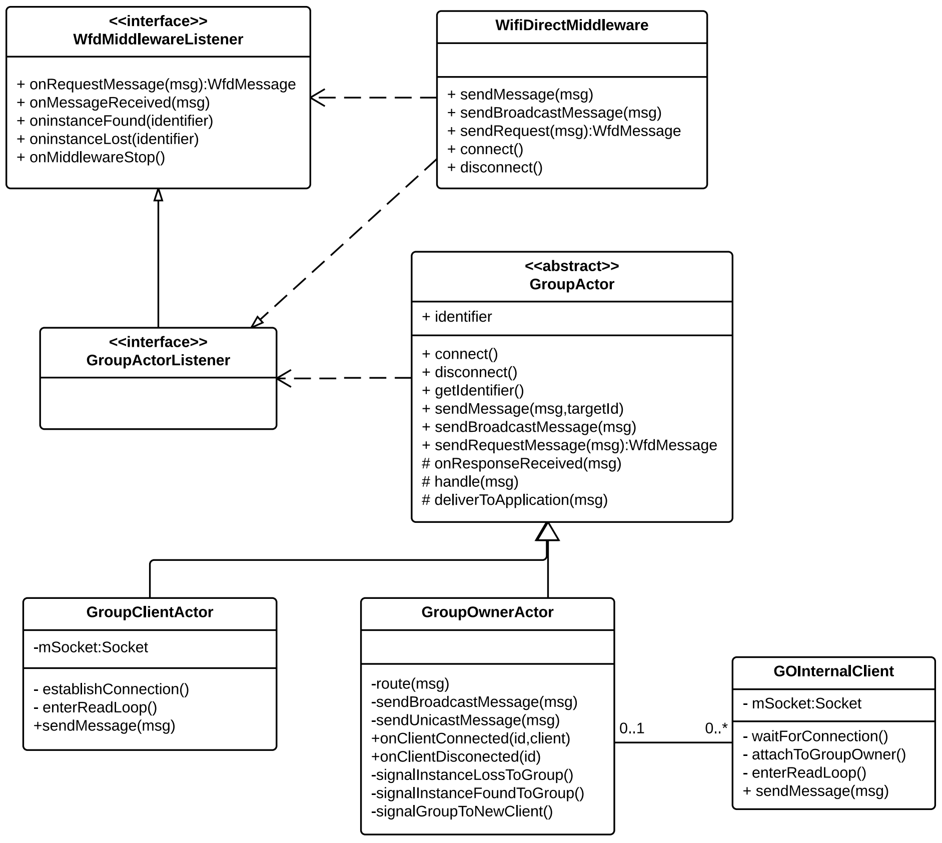


Figure 4.12: Class diagram of the Wi-Fi Direct middleware

group owner, that include the group management as well as the routing of messages within the group.

A `GOInternalClient` represents the server side connection of a `GroupClientActor`. They are created by the `GroupOwnerActor` that accepts the incoming connections and uses them to send messages and to be notified of received ones. A `GOInternalClient` has to monitor the connections and notify the group owner when a peer leaves the network.

Finally, `WifiDirectMiddleware` class provides a coordination layer that handles the discovery of peer, the creation of the P2P Group and mediates the communication to the upper layer of the system. In particular, this last function is made with a series of adapters whose aim is to adapt the Wi-Fi Direct middleware interface to the ones required by SPF. These components are not shown in figure since they resemble the ones used for the AllJoyn implementation.

Chapter 5

Results and evaluation

Software is getting slower more rapidly than hardware becomes faster.

Wirth's Law

This chapter discusses the experimental realizations and the evaluation of this work. The first section presents two of the applications we have developed using SPF. On the basis of the scenarios outlined in Chapter 3, a thorough description of these applications helps to understand how the functionality of SPF can be used to build social proximity applications. The second section provides an assessment on the code quality achievable by using SPF compared to other communication middleware. Here we compare four different configuration of a social proximity application on the basis of code quality metrics and architectural components. It helps to understand how a higher level of abstraction can ease the development of proximity oriented application. Finally, the third section analyzes the impact and limitations of the inter-process architecture of SPF. To do this, we compare the response time of a network transaction according to different communication means and payload sizes.

5.1 Samples of applications

5.1.1 SPFChatDemo

SPFChatDemo is an Android application that enables proximity-based social interactions, according to the SNiP scenario described in section 3.1.1. In particular, SPFChatDemo offers the following functions:

People discovery : The application enables the discovery of people in proximity, displaying the list of found results. Users can also specify a set of properties that the profile of results must match. Once a new person is discovered, the app allows further social interactions.

Profile Visualization : Users can view the profile of other people in proximity.

Greeting : Users in proximity can exchange *greets*, instant interactions that results in a toast notification displayed on the phone of the target, if the app is in use, or in a system notification otherwise.

Chat : A user can start a conversation in real time with a person discovered in proximity by sending a new text message. Incoming messages are notified by means of system notifications.

These functions have been implemented relying on the capabilities offered by the SPF Framework. The first function, People Discovery, is built upon the Search API. In particular, the application creates a SPF query that matches all SPF instances that have SPFChatDemo installed. The user can further customize this query by adding property and tag parameters through the UI of the application. When the user starts the discovery process, the application registers a search in SPF providing the query, a fixed search configuration and a callback. This callback is used to update the list of discovered instances visible to the user: each entry displays the person's name, and two buttons, one for the greet interaction, and one to send a chat message. If no people has been discovered once the process finishes, a message is displayed to the user.

The user can display the profile of a person by clicking on its entry in the list of discovered people. This features displays a predefined subset of the available profile fields: to view the full profile of a person, the user can open it in the SPF Provider application. The values of profile fields are obtained from the remote instance using the SPF Profile API. Therefore, the visibility of field values depends on the clearance set by the remote user, thus not all fields may be accessible.

The Greeting and Chat functions are built upon the Services API using activities. In particular, SPFChatDemo uses the inter-operable verbs `greet` and `message` to exchange greets and messages, respectively. The consumption of these activities is implemented by `ProximityService`, a SPF Service with two methods annotated as activity consumers, one for greets, and one for messages.

The implementation of the service is defined by means of the `ProximityServiceImpl` class. It features a pluggable strategy that enables the customization of the business logic that handles incoming activities. Incoming greets are directly dispatched to the currently selected strategy, while chat messages are persisted in a database, then dispatched to the strategy.

The default reaction strategy dispatches a system notification every time a greet or chat message is received. In this way, if the application is not running and SPF binds to the service implementation to deliver an activity, the default strategy is used, thus resulting in a system notification. On the other hand, when the application is active, the default strategies are overridden by components of the user interface, in order to provide real-time updates. For example, the list of chat conversation and the list of messages in a conversation are automatically updated when a message is received.

5.1.2 SPFCouponing

SPFCouponing is a couple of Android applications that enable the proximity-based distribution of digital coupons tailored to the social profile of shop customers. These applications realize the Targeted Advertising scenario, described in section 3.1.1. These applications are based on the concept of *Coupon*, which is a special offer for products of a given category, gifted to a customer by a shop owner. Each coupon features a title, a text message and a photo. To determine if a coupon is tailored to a given customer, the Provider application matches the public social profile with the product category of the coupon: if the profile contains the category, then probably the customer is interested in related coupons. For example, if the profile of a user contains the "Smartphones" term, then he probably is interested in receiving coupons related to novel Android devices.

The first application, named *SPFCouponingProvider*, is targeted at shop owners and provides the following functions:

- **Coupon Configuration**: Shop owners can configure the coupons that are delivered to interested clients.
- **Category Advertising**: Shop owners can select which product categories should be added to the social profile and thus advertised to clients.
- **Welcome message**: Shop owners can configure a text message that is delivered to all nearby customers upon the first visit.

The second application, named *SPFCouponingClient*, is targeted at customers and interacts with SPFCouponingProvider to receive coupons targeted to the user. The functions offered by this application are:

- **Coupon browsing**: Customers can browse the list of coupons received from nearby shops.
- **Category Selection**: Customers can select which product categories they are interested in: in this way, they will receive a notification every time a shop selling a favorite category is nearby. Customers can also opt-in to receive coupons for each category they select.

The interaction between the Provider and the Client applications is implemented leveraging only on the capabilities offered by SPF; in particular, the Notification, Services and Profile API are used.

Coupon Delivery

As previously described, digital coupons are created by shop owners using the Provider application. When the owner defines a new coupon, the application registers a new SPF trigger to react to customers interested in the coupon, as described in figure 5.1. The query of this trigger matches SPF instances that have the client application installed, and that contain the coupon category name as *tag*. The action specified by the trigger is an IntentAction, whose intent activates the delivery of the coupon to the client application. Once the Trigger is registered, the provider application stores the Coupon information, including the coupon detail (title, text, photo and category) and the trigger ID, into a database.

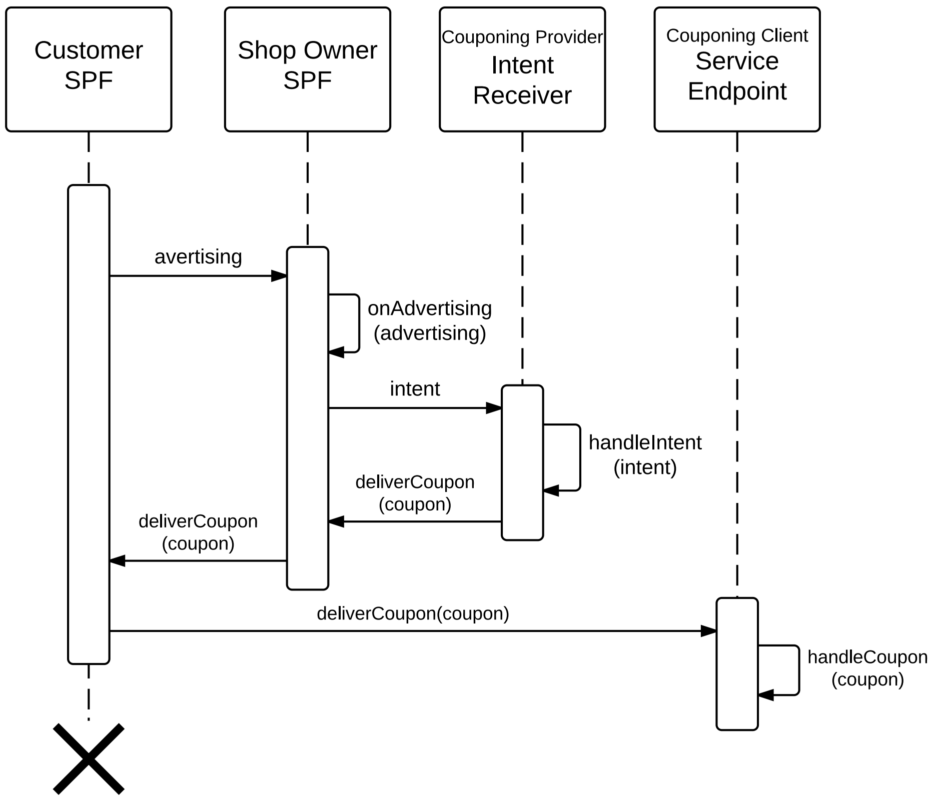


Figure 5.1: Coupon delivery

The delivery of coupons to client applications is implemented by an Android *IntentReceiver*, activated by the intents dispatched by SPF when coupon triggers are fired. These intents contain the id of the trigger that caused the intent: with this piece of information, the receiver obtains the original coupon from the database. The actual delivery of the coupon is implemented by means of an SPF service, called `Coupon-DeliveryService`, which is exposed by client applications. Using the identifier of the remote instance contained in the intent, the receiver obtains a reference to the remote SPF instance and executes the service method to deliver the coupon to the client.

The CouponingDeliveryService, exposed by Client applications, contains only one method, named `deliverCoupon`. The service implementation features a pluggable strategy approach, similar to the one of SPFChatDemo. After received coupons are saved in a database, the implementation delegates the reaction to a strategy that can be set from other application components. When no external strategy is set, the default one is executed, which causes a system notification to be displayed to the user. External behaviors, on the other hand, are set from GUI components to implement in-app reaction to new coupons. In particular, if the user is currently browsing available coupons, the list will update automatically.

Shop Notification

As previously described, the client application lets customers select the product categories in which they are interested. After a category is added, the user will receive notifications when a shop offering a favorite category is nearby and, possibly, coupons for selected product categories. Since the coupon delivery system is based on triggers, when the user selects to receive coupons for a given category, the provider application adds it to the *interests* field of the social profile.

The shop matching behavior is based on SPF triggers as well. When the customer adds a new category to their favorites, the application creates a new SPF trigger, whose query matches SPF instances that have the provider application installed, and that contain the product category as *tag*. Similarly to coupon delivery, the trigger specifies an intent action whose intents activate an `IntentReceiver`. In this case, the receiver is designed to dispatch system notifications when intents are received. These notifications, once activated, display the social profile of the shop.

Welcome Message

The welcome message function allows shop owners to define a message, composed by a title and a body, that is dispatched to customers the first time they are nearby the shop. The delivery of messages rely on the SPF Notification API as well: when the welcome message is activated, the Provider application registers a SPF trigger that matches all instances with the Client application installed. The action of the trigger is an `ActionSendNotification` that dispatches the welcome message to the remote instance within the SPF Provider application. The trigger is configured as one-shot, so that it is fired only once for each customer.

5.2 Code quality

One of the main objectives of SPF is easing the development of social proximity applications. The complexities of the development of such services arise from the intricacies of the underlying networking technologies and the absence of high level abstractions. Moreover, specific constraints comes from the peculiar nature of mobile software components, that add to the developer the burden of dealing with background components, multi-threading and synchronization issues, as well as a potentially complex management of resources. All these issues force the developer to design the whole infrastructure, spending his efforts in the engineering of the communication layers, rather than focusing on the main functions of the application.

To understand and assess the effectiveness of SPF, we decided to analyze the code-base of a proximity based chat implemented in four different configuration that use or do not use the SPF. The first comparison we propose compares the Wi-Fi Direct chat, bundled in the Android SDK samples, against the same application but with the networking layer substituted by SPF. The second analysis is between the AllJoyn chat provided within the SDK and a SPF Chat sample.

To evaluate the code of the different applications we used SonarQube[1], an open source platform to measure code quality. We focused on two important metrics: the overall lines of code, and the cyclomatic complexity, which counts the number of different paths in the source code and provides a quantitative measure of the complexity of the code. A high cyclomatic complexity is not an issue per se, as it depends on the size of the program. However, it can be used to compare two applications that implement the same functionality, since a less complex implementation is easier to debug and maintain.

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

Wi-Fi Direct and SPF

The Wi-Fi Direct chat sample we considered is the one provided within the Android SDK[2]. It is a simple app that allows two devices to exchange messages. When the app is launched, it uses the service discovery capabilities of Wi-Fi Direct to provide a list of available peers. This implies the local registration of the service and the start of a discovery procedure. Once one of the user has selected a service, a Wi-Fi Direct connection is performed.

According to the result of the negotiation, two different subclasses of Thread are started. If the device is a group owner, it instantiates a `GroupOwnerSocketHandler` that holds a pool of thread for incoming socket connections. If the device is a standard peer, the `ClientSocketHandler` is used to establish a socket connection and then running another thread to read from the socket. The component for writing and reading from the socket is shared between the two cases and it is called `ChatManager`. The decoupling between the IO threads and the UI one is implemented with an Android Handler, that allows to deliver messages in the main thread.

To substitute Wi-Fi Direct with SPF, we defined a simple SPF service for the reception of textual messages and used the SPF Search API to perform the discovery of peer in proximity.

Tables 5.1 and 5.2 shows respectively the data about the size of the code and its cyclomatic complexity. As expected the introduction of SPF and its higher level of abstraction, led to a considerable decrease in the overall cyclomatic complexity and code-base size.

	WfdChat	**WfdChat on SPF**
Lines of Code	619	301
№ of Files	8	5
№ of Classes	12	9
№ of Functions	32	32

Table 5.1: Wi-Fi Direct Chat code-base size

[2]samples/android-17/WiFiDirectServiceDiscovery

Complexity	WfdChat	WfdChat on SPF
cc/function	3	1.6
cc/class	8.1	5.7
cc/file	12.1	10.2
cc total	97	51

Table 5.2: Wi-Fi Direct Chat cyclomatic complexity

AllJoyn and SPF

For the second analysis we considered the chat bundled in the Alljoyn SDK as a sample. Since its logic structure is very tied to the capabilities of the AllJoyn framework, we were not able to provide a such precise comparison as in the Wi-Fi Direct case: in fact we compared the AllJoyn demo app with an our application called *SPFChat* that provides a similar functionality. Despite that, as we will discuss later, it was still interesting to compare the SPF design with another similar high-level framework.

AllJoynChat allows a group of devices to exchange messages between each other. A user may decide to create a *channel*, which represent a standard chat room. To create a channel the application register and advertise a well-known name which is composed by a predefined prefix and a suffix that depends on the name that the user assigned to the channel. Remote clients may use the same application to search for available channels and join a chat room. The concept of group is implemented by means of multi-point sessions and bus signals. Each user of the application has to register a bus object whose interface declares a single bus signal that allows the broadcasting of textual messages. The advertised well-known name is used as an endpoint to create multi-point session. Once a user has joined a session, i.e., a channel, he can receive and send bus signals that are delivered to all the users within the established session. An Android foreground Service is used to keep the connections opened and to hold the AllJoyn resources.

Differently from AllJoynChat, SPFChat does not allow the creation of chat rooms, but offers a more usual interaction. Users of SPFChat may discover other peers in proximity and start a conversation. As for the Wi-Fi Direct case, we used SPF search and one SPF service to implement respectively the discovery and the chat function. Moreover, conversations are stored persistently in a SQLite database.

Code size and cyclomatic complexity are shown respectively in tables 5.3 and 5.4.

	AllJoynChat	**SPFChat**
Lines of Code	1392	712
№ of File	9	8
№ of Classes	16	15
№ of Functions	95	63

Table 5.3: SPFChat and AllJoynChat code-base size

Complexity	**AllJoynChat**	**SPFChat**
cc/function	2.3	1.9
cc/class	14.8	8.1
cc/file	26.2	15.3
cc total	236	122

Table 5.4: SPFChat and AllJoynChat code-base cyclomatic complexity

The major differences between the two application do not regard the different complexities of the offered functions. In fact, the programming model for group communication offered by AllJoyn is almost the same of the one offered by spf services, except for the concept of multi-point session. The real difference about AllJoynChat is inherent in how resources are handled. These are mainly managed by the foreground Android Service in a class called `AllJoynService`, that holds a reference to the AllJoyn bus attachment along with the state of the network. To allow the application to access the middleware, it offers a complex multi-threading structure that handles operations and asynchronous events. These characterize the life-cycle and the interactions between the bus attachment and its application. On the other hand, SPF takes care of all these problems by providing a simpler abstraction that automatically handle the network resources and, more important, is tied to the life-cycle of the Android components. As result, SPFChat does not need a foreground service, since this feature is already supported by the SPF framework.

5.3 Performance

We then evaluated how the introduction of the SPF affected the performance of the apps that rely on it. The experiments aimed at collecting the response times of different implementations of the same service/app that differ for the adopted communication layer.

More in detail, we created a ping service that sends requests and waits for acknowledgments from the receiver. The different sizes of the payload were obtained by providing a string obtained by concatenating a varying number of characters. We collected the response times on two Samsung Galaxy S4 smartphones: one acting as server and the other one as client. The client phone was in charge of performing service invocations, one after another, by varying the size of the payload. The experiments considered five different configurations that differed in the used communication middleware: SPF- AllJoyn, SPF-WiFi Direct, Alloyn alone, WiFi Direct alone, and a local-only implementation, that is, a configuration that only used inter-process communications on the same device without remote invocations.

Figure 5.2 shows the results of these experiments. If we compare the response time of a local transaction with the one of a remote service invocation, as expected, the impact of inter-process communication is negligible with respect to the dimension of a network transaction. If we analyze the results with and without the SPF, we discover that the current implementations of the framework causes extra delays on the response time of up to 50ms for the Wi-Fi Direct implementation and up to 200ms for the AllJoyn implementation. Tool Traceview by Android helped us understand that the delays are mainly due to the serialization and deserialization of message contents. Most of the time spent in an SPF invocation is spent on parsing the received message, which is serialized as a JSON object. This is caused by the internal garbage collection that depends on the implementation of the *gson*[3] library that we have used for the serialization. The difference between the response times of the two configurations with the SPF is

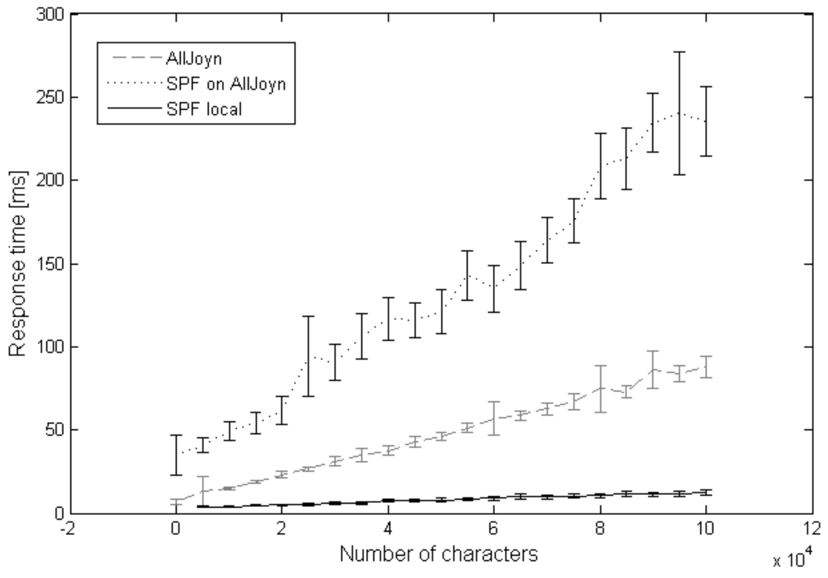
[3]`https://code.google.com/p/google-gson/`

the following: the WiFi Direct- based implementation performs the JSON serialization and deserialization within the middleware, while the AllJoyn-based implementation does it on top of the middleware.

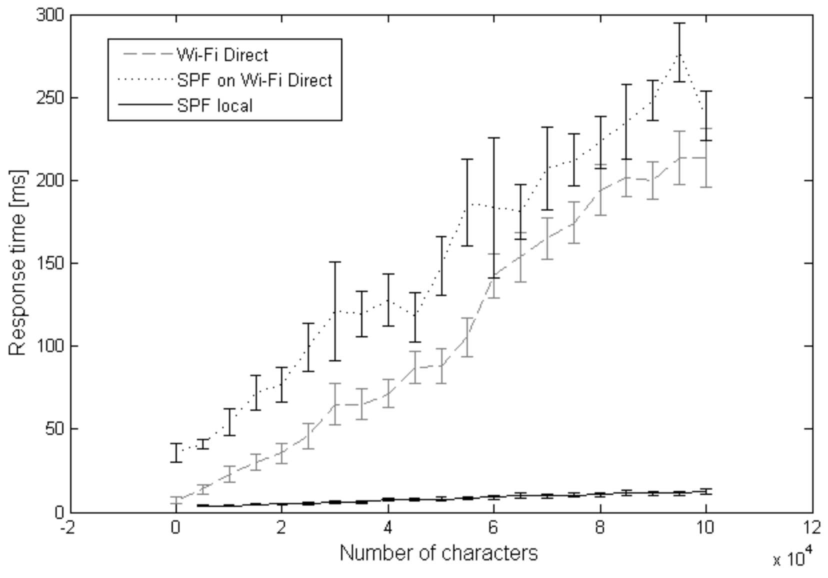
Based on the same ping service described above, we also performed an additional experiment to evaluate how the framework is able to handle concurrent remote invocations that are required especially in scenarios where multiple user devices access at the same time the same service: for example, one of those provided by a device associated with a smart space, as discussed in the scenarios of chapter 3. In this case, we found that the framework suffers scalability problems due to intrinsic limitations imposed by the Android IPC (Inter Process Communication) framework. Indeed, as described in the API reference documentation[4], the arguments and return value of a remote procedure call are stored in a buffer that has a limited fixed size of 1Mb; this buffer is shared among all the transactions in progress for the process handling the remote invocations. This means that even if an individual transaction has a moderate size, it may fail because of the load of the system. The problem can be overcome by synchronizing remote invocations through a queue. For the same reason, currently a single remote invocation must have a payload that does not exceed the size of the buffer imposed by Android.

To understand the impact of this limitation we analyzed the size of social content exchanged in the web. The usual messages and pictures used on the web and on traditional social networks do not suffer this problem. Therefore is possible to exchange small profile picture and even thumbnails to be shown in an activity feed. However the problem occurs when the payload contains large amount of data, such as high-resolution pictures or videos. As future work we will address the performance overhead due to serialization and we will extend the framework by providing dedicated APIs for large data transfers.

[4]`http://developer.android.com/reference/android/os/TransactionTooLargeException.html`



(a) SPF and AllJoyn



(b) SPF and Wi-Fi Direct

Figure 5.2: Response time comparison with .95 confidence level

Chapter 6

Conclusions and future works

Computers are useless
they can only give you answers.

Pablo Picasso

Nowadays, the interest of industry for IoT scenarios is leading the efforts towards the creation of novel technologies to enable the interaction with smart objects and environments. Usually these technologies neglect the social identity of the user, because they focus more on networking than on providing high-level frameworks. However, the social element can still be exploited to create new services that are tailored to the user and support novel interactions with the environment and its elements. These concepts, presented in Chapter 3, can be synthesized in the term *social smart space*. A social smart space is a physical space where members, both humans and smart objects, can exploit services provided by the others in a technology-mediated way. The social identity is then used to personalize the services and adapt the behavior of the smart space according to the habits and the preferences of its users.

When trying to design such a system, the world of web services and social networks became the natural reference model. The different constraints imposed by the device to device communication, the proximity scenario, and the more personal nature of mobile devices led us to define a new infrastructure that could fit these particular requirements.

The Social Proximity Framework is the result of this work: a software solution for the creation of social smart spaces where services are tied to the user identity, providing means for novel types of interaction without the need of an external infrastructure. The main features of SPF can be summarized in three points:

- high-level abstractions over the networking layers, to ease the development of social proximity applications;
- facilities for the development and integration of the most common social functions;
- control on behalf of the user, that can manage data and services hold within the framework.

Developers may access the framework by means of a dedicated software library. Users may control and personalize the framework resources through a SPF Provider application.

The Android inter-process communication framework allowed us to design a centralized architecture where network resources and data are shared between all the applications that use SPF. From the user perspective, this integration between different apps is seamless. Even the developer does not notice any issues when trying to access the SPF, since all the communication is mediated and simplified by the framework library.

SPF relies on an abstract communication layer that provides the underlying networking functions. This allowed us to test the framework against two different network infrastructures based respectively on AllJoyn and Wi-Fi Direct. This has shown some problematics and weaknesses of the model that could prevent the adoption of a different technology. More in detail, these issues regard the discovery functions and the scalability of SPF. In fact, according to the capabilities of the network technology we may have different means to perform multicast and connection-less communication. Among the emerging technologies, the one that is the most promising for the SPF is LTE Direct, discussed in Section 2.1.4. Since SPF shares with LTE Direct the same application scenarios, it would be a good candidate as a model to shape social proximity services and present them to the user in a coherent and comprehensible way. Another interesting approach is trying to overcome the limitations we encountered with Wi-Fi Direct by means of other technologies that could offer more flexible functions for implementing the discovery; these can be Bluetooth or LTE Direct itself.

The second important future work regards the porting of the solution on different operating systems. The success of the Android implementation is based on the underlying IPC infrastructure. Other operating systems, like iOS, does not offer such a flexible solution. In this case the whole architecture has to be redesigned according to the available features.

The number of smart devices in the world is steadily increasing, and so is the average number of devices per person. The recent introduction of wearable smart devices has further increased the possible interaction scenarios. Since SPF focuses on the user identity, it should take into account multiple devices owned by a single user and present a consistent interaction that is person-oriented and not device-oriented. A possible approach would be the introduction of a synchronization process that bootstraps a SPF instance on a new device using the data from another device of the user. Each instance that belongs to the same user may advertise the same digital identity, and a different device profile. However, this approach requires a constant synchronization of the information on different instances: a modification of profile field that happens on a device should be propagated to all other synchronized devices.

However, the synchronization among different devices is prone to security risks and also clashes with the current infrastructure-less architecture of SPF. Every modification made to the profile on a device could be propagated with another device only when they can see each other in proximity. Concurrent modifications are also more complex to solve without a central coordinator. Thus, a possible extension to the framework is the introduction of an external infrastructure that integrates with the pure peer-to-peer nature of SPF. This infrastructure would provide identity management function, thus coordinating multiple SPF instances on devices belonging to the same user. Also, the profile information, and each change performed, could be synchronized with the remote infrastructure.

Another possible future work is the introduction of semantic techniques to improve the analysis of user profiles and their matching against search and advertising queries. In particular, category-matching algorithms would enable a more powerful profile discovery. Currently, an application can discover nearby people interested in smartphones only by means of a tag search for profiles containing a specific term, e.g, smartphone. On the other hand, category-matching techniques would link to smartphones those profiles that contain related terms, like “Android”.

SPF may also be extended towards the world of the web social networks. A first integration step can be a bootstrap process that initializes the profile information of a new SPF instance using an existing on-line social network as source. In this way, the user experience would benefit as setup time is reduced. However, this scenario requires an internet connection and the development of multiple connectors to interact with the public API of each supported service.

A further step in the integration with on-line social networks would be the compliance with the SNeW specification, introduced in Section 2.2.3. In particular, SPF would take the role of *SNeW client*, allowing the interaction with compatible social network. This would enable SPF users to interact with people not only in direct proximity, but also with those on supported social networks. This integration would also enable the cross-device synchronization scenario previously described, without the burden of a proprietary external infrastructure, as its functions are implemented by existing social networks.

Appendix A

Guide to SPF Library

A.1 Introduction

This document is a guide for using the *Social Proximity Framework* (*SPF*) to write Android applications. More concrete samples can be found in the official repository of the project: `http://github.com/deib-polimi/SPF`.

A.2 Overview

`SPF Lib` is made up by modular components that provide access to `SPF` functions. Loading a component requires an asynchronous operation during which a connection to the framework is established. Among these components there is one of them which is fundamental for the networking functions and it is called `SPF`. Once this component has been loaded, the framework activates the proximity middleware and thus the device becomes visible to remote users until `SPF` is released. Listing A.1 shows how this component can be loaded and used to access the framework proximity services. The other components follow a similar pattern and are discussed in the following sections.

Listing A.1: Loading SPF connection.

```
SPF.load(context,new SPF.Callback(){
    void onServiceReady(SPF spfConnection){
        SPFSearch search = spf.getComponent(SPF.SEARCH);
        //Do your work here...
        //...and disconnect when finished
        spf.disconnect();
    }

    void onDisconnect(){
        //disconnection from the framework
        //the connection is not valid anymore
    }

    void onError(SPFError err){
         //handle errors here
    }
});
```

External applications must ask for permissions according to the services they want to use. Listing A.2 shows how to use the `SPFPermissionManager` class to declare the needed permissions. Declaration must be performed before accessing the framework e.g. in your `Application` class or in the `onCreate` method of your `MainActivity`.

Listing A.2: Declaring permissions.

```
SPFPermissionManager.requirePermission(
    Permission.SEARCH_SERVICE,
    Permission.READ_LOCAL_PROFILE,
    Permission.REGISTER_SERVICES);
```

According to the SPF provider application you want access, you should provide additional configuration settings by means of the SPFInfo class. These parameters are used to create the android `ComponentName` that identifies the `Service` which offers the access to SPF.

Listing A.3: Configuring SPFInfo.

```
SPFInfo.PACKAGE_NAME = "my.spf.frontend.app";
SPFInfo.CLASS_NAME = "my.custom.spf.service.SPFService";
```

The configuration is required only if you are not using the front-end application provided with the framework. Follows a description of the parameters to be set:

PACKAGE_NAME The package name of the SPF provider application. Use the one of the front end application, by default it is set to spf official front-end i.e. `it.polimi.spf.app`.

CLASS_NAME The class name of the service that offers SPF interfaces. By default it is set to `it.polimi.spf.framework.local-.SPFService`, modify the constant according to the service registered in the front-end application.

A.3 SPFSearch API

This section describes how an application can discover remote instances of *SPF* according to customizable queries. A search works by broadcasting a query signal to remote devices in proximity; if it matches their user profiles, a response is sent back along with some basic information about the user. To access SPFSearch API, you must ask for `SEARCH_SERVICE` permission.

A.3.1 Defining a query

SPF Search requires the creation of a `SPFQuery` object, that can be created with a builder defined in its class. A query can be configured with different types of parameters:

Profile field value returns all the users that contain the value specified in the query for the given profile field.

Tag returns all the users that contain the specified word in any profile fields.

App Identifier returns the users that installed the given application. The app identifier is the application package name.

The query is satisfied if all the provided parameters are matched; thus, if you want to search for all the women related to a tag "Android", you should specify "female" as value of gender profile field and "android" as a tag. Providing an empty query returns all the instances of *SPF* in proximity. Listing A.4 shows an example.

Listing A.4: Building a query

```
SPFQuery query = new SPFQuery.Builder()
    .setAppIdentifier("com.spf.demo.search")
    .setProfileField(ProfileField.GENDER,"female")
    .setTag("android")
    .build();
```

A.3.2 Starting a search

Once you have created a query, you have to define a `SearchDescriptor`, which allows configure the behavior of the search operation. In detail, the `SearchDescriptor` is made up by the `SPFQuery` object and two additional parameters: the number of signals to be sent, and the time interval between them, the latter measured in milliseconds. Finally, you must acquire a connection to the *SPF* proximity service and ask for `SPFSearch`. Listing A.5 provides an example.

Listing A.5: Starting a search

```
long interval = 5000; // 5 seconds
int numberOfSignals = 5;
SPFSearchDescriptor descriptor =
    new SPFSearchDescriptor(interval, numberOfSignals, query);
SPFSearch search = spfConnection.getComponent(SPF.SEARCH);
search.startSearch(SEARCH_TAG, descriptor, mSearchCallback);
```

The `startSearch` method requires two more parameters. The String `SEARCH_TAG` is simply a string that identifies the request and can be used to control a running query; when you provide an identifier already owned by another running request, it is automatically stopped by the framework. `SPFSearchCallback` contains the callbacks to be executed when the framework delivers the results to your application:

onSearchStart notifies that the search has begun. This is the right moment to show the user a progress bar.

onPersonFound is called when a search result is collected. It delivers information about the discovered user in a `SPFPerson` object. The method `getBaseInfo()` allows to retrieve the identifier and the name of the user.

onPersonLost is called when a previously found user has been lost.

onSearchStop is called when the search operation has stopped and no further result will be delivered.

onSearchError is called when an error occurs while executing the search.

If you want to retrieve a stub of a spf instance whose identifier is known, you should use the `lookup` method; if the remote instance is reachable, it returns a `SPFPerson` object that can be used with other *SPF* services.

A.4 SPFProfile API

SPF allows applications to read and write on a shared user profile. To access these data you must load `SPFLocalProfile`, which requires `READ_LOCAL_PROFILE` and `WRITE_LOCAL_PROFILE` permissions, according to the needed use. Be aware that the two are independent, thus if your application wants to read and write the profile, it must require both the permissions.

A.4.1 Reading local profile

Once the component is loaded, you can read or write the profile specifying the values you want to access. The class `ProfileField` holds the definitions of profile fields. Listing A.6 shows how a read operation can be performed.

Listing A.6: Reading from local profile.

```
SPFLocalProfile.load(context, new SPFLocalProfile.Callback() {
    public void onServiceReady(SPFLocalProfile spfLocalProfile) {
        ProfileFieldContainer container = spfLocalProfile.getValueBulk(
            ProfileField.IDENTIFIER,
            ProfileField.DISPLAY_NAME,
            ProfileField.PHOTO,
            ProfileField.BIRTHDAY );

        String uid = container.getFieldValue(ProfileField.IDENTIFIER);
        String name = container.getFieldValue(ProfileField.DISPLAY_NAME);
        Bitmap profilePic = container.getFieldValue(ProfileField.PHOTO);
        Date birthday = container.getFieldValue(ProfileField.BIRTHDAY);
        spfLocalProfile.disconnect();
    }

    public void onError(SPFError spfError) {...}

    public void onDisconnect() {...}
});
```

A.4.2 Writing local profile

Writing profile fields works in a similar way. You have to use a `ProfileFieldContainer` by creating a new one, or by reusing the one that is returned as result of a read operation. Old values are overwritten by the new ones contained in the data structure[1].

Listing A.7: Writing a profile fields

```
ProfileFieldContainer container = new ProfileFieldContainer();
Date birthDate = new Date();
String[] emails = new String[]{"an.email@email.com","another@email.com"};
container.setFieldValue(ProfileField.BIRTHDAY, birthDate);
container.setFieldValue(ProfileField.EMAILS, emails);
spfLocalProfile.setValueBulk(container);
```

A.4.3 Reading remote profile

Application can read profiles of remote users; while the interface is similar to the one of the local profile, accessing remote devices requires the loading of the `SPF` component as described in section A.2. Listing A.8 shows the three steps procedure needed to access a remote profile: obtaining the `SPFRemoteProfile` component, creating a stub from a `SPFPerson` instance[2] and finally, asking for the desired profile fields.

Listing A.8: Reading a remote profile

```
SPFRemoteProfile remPr = spfConnection.getComponent(REMOTE_PROFILE);
remPr.getProfileOf(spfPerson).getValueBulk(...);
```

A.5 SPFService API

This section describe and provides examples about SPF service API. According to the intended use, your application should require the following permissions: `REGISTER_SERVICES`, `EXECUTE_LOCAL_SERVICES`, `EXECUTE_REMOTE_SERVICES`.

[1]`ProfileFieldContainer` keeps track of the modified values: only the modified fields are actually written. Thus you can use this data structure as a model for profile editing activities. For more details see the class documentation.

[2]`SPFPerson` instances can be obtained from a search or a lookup. See section A.3 for details.

A.5.1 Defining a service

The `@ServiceInterface` annotation can be used on a standard Java interface to define and configure a *SPF Service.* In the annotation you must specify:

app The identifier of the application (i.e. its package name).

name The name of the service that you are defining.

description A textual description of the service to be shown to the user. Should contain useful information to let the user recognize the service.

version The version of the service.

Listing A.9: Defining a service interface.

```
package com.example.chat;
import it.polimi.spf.lib.services.ServiceInterface;

@ServiceInterface(
    app = "com.example.chat",
    name = "SPFChatDemo Proximity",
    description = "Service that allows users to communicate",
    version = "0.1",
)
public interface ProximityService {

    void sendMessage(String senderId, String message)
        throws ServiceInvocationException;

    void sendPoke(String senderId)
        throws ServiceInvocationException;

}
```

Listing A.9 shows an example of how the annotation should be used. To provide an implementation of the service you have to create a class that extends `SPFServiceEndpoint` and implements the previously defined interface. Pay attention to declare the `ServiceInvocationException` in each remote method signature; the omission will prevent to recover from errors and will cause a runtime exception when one of these occurs.

A.5.2 Registering a service

Once the service interface and its implementation are defined, the application needs to register the service. To do so, you must load `SPFServiceRegistry` as shown in Listing A.10.

Listing A.10: Registering a service.

```
SPFServiceRegistry.load(context, new SPFServiceRegistry.Callback() {

    public void onServiceReady(SPFServiceRegistry serviceRegistry) {
        serviceRegistry.registerService(
            ProximityService.class, ProximityServiceImpl.class);
        serviceRegistry.disconnect();
    }

    public void onError(SPFError spfError) {...}

    public void onDisconnect() {...}
});
```

Eventually, you have to register your subclass of `SPFServiceEndpoint`[3] in the manifest file of your application.

A.5.3 Executing a service

Executing a service is a two step operation that requires the creation of a stub and the method invocation. `SPFServiceExecutor` is the component that allows you to create the service stub given the instance of `SPFPerson` referring to the remote user, the interface of the service that you want to invoke, and eventually the class loader. Listing A.11 shows how the previously defined service can be executed.

Listing A.11: Executing a service.

```
SPFServiceExecutor executor = spf.getComponent(SPF.SERVICE_EXECUTION);
try {
    ProximityService myService = executor.createStub(
        spfPerson, ProximityService.class, getClassLoader());
    myservice.sendMessage(myIdentifier, message);
} catch (ServiceInvocationException e) {
    Log.e(TAG, "Cannot send message", e);
}
```

[3]`SPFServiceEndpoint` is an abstract class that extends a common Android `Service`.

A.5.4 Supported data types

All primitive Java types are supported, along with collections and arrays. Moreover, the library supports serialization and deserialization of custom classes and their respective arrays, if they do not have circular references. Limitations on the request size depends on the proximity middleware in use[4] and on the Android Binder buffer used for inter-process communication[5].

A.6 SPFNotification API

`SPF` allows users to advertise selected fields of their own profile. This functionality can be configured from the framework front-end, and offers a more efficient solution for long running searches that work in background. The actual implementation depends on the networking middleware in use, but basically it consists in broadcasting a set of selected profile fields that is received by other `SPF` instances. External applications may react to the reception of these information by defining triggers. To access notification services, applications must declare `NOTIFICATION_SERVICES` permission.

A.6.1 Triggers and actions

`SPFTrigger` is the abstraction that represents the "if. . . then. . . " rule, and can be used to react to the reception of an advertised profile. A trigger is made up by two components: a query and an action. The query is the same object defined in section A.3, and it allows to define searches for specific profile fields as well as more general queries. There exist two types of action that can be declared in a trigger:

`SPFActionIntent` broadcasts an Android `Intent` that can be received from the application that registered the trigger; This `SPFAction` requires an action name for the `Intent` and provides information about the remote `SPF` instance that activated the trigger.

[4]E.g. on AllJoyn the limit size is 100Kb.

[5]It has a limited fixed size of 1Mb, shared by all the transactions in progress for the process. More details can be found on Android documentation following this link: `http://developer.android.com/reference/android/os/TransactionTooLargeException.html`.

`SPFActionMessage` sends a message to the remote user that has activated the trigger. The framework front-end will notify the user about the received message.

Moreover, applications can specify whether a trigger is to be activated multiple times by the same remote instance or not. In the first case you should provide a `sleepPeriod`, a time interval in milliseconds during which the trigger is not activated on a already targeted user.

Listing A.12 shows the creation of a trigger with a `SPFActionIntent` to count and log all the encountered instances of `SPF`. Notice that in this case no `sleepPeriod` is specified: in this way the intent is broadcast only once for each individual instance. Finally listing A.13 shows the implementation of a `BroadcastReceiver` that uses the information provided by the framework by mean of the `Intent`.

Listing A.12: Trigger for logging instances.

```
SPFQuery query = new SPFQuery.Builder().build();
SPFActionIntent action = new SPFActionIntent("com.trigger.example.COUNTER");
String name = "People counter";
SPFTrigger trigger = new SPFTrigger(name, query, action);
```

Listing A.13: BroadcastReceiver for logging instances.

```
public class PeopleCounter extends BroadcastReceiver{

    public void onReceive(Context context, Intent intent) {
        if (intent.getAction().equals("com.trigger.example.COUNTER")){
            String triggerName = intent
                .getStringExtra(SPFActionIntent.ARG_STRING_TRIGGER_NAME);
            String targetId = intent
                .getStringExtra(SPFActionIntent.ARG_STRING_TARGET);
            String displayname = intent
                .getStringExtra(SPFActionIntent.ARG_STRING_DISPLAY_NAME);
            Log.d(TAG, "Found user: " + displayName + " id: " + targetId);
    }

}
```

Listing A.14 shows an example of a welcome message sent once in a day. Here a `sleepPeriod` is set and a `SPFActionMessage` is used.

Listing A.14: Trigger for welcome messages.

```
SPFQuery query = new SPFQuery.Builder().build();
String object = "Welcome";
String message = "Welcome in this beautiful Social Smart Space!";
SPFActionMessage action = new SPFActionMessage(title,message);
SPFTrigger trigger = new SPFTrigger(name, query, action, DAY_MILLIS);
```

A.6.2 Registering triggers

To access `SPF` notification services you have to load `SPFNotification` component. Once you have obtained an instance of this object you can call the methods that allows to register and modify the triggers.

`saveTrigger(SPFTrigger trigger)` registers the trigger given as parameter and assigns an id.

`listTriggers()` returns the list of trigger that have been registered by the application.

`getTrigger(long id)` returns the trigger with the specified id

`deleteTrigger(long id)` deletes the trigger with the specified id

`deleteAllTriggers()` deletes all the triggers that have been registered by the application.

Listing A.15: Reading saved triggers.

```
SPFNotification.load(context,new SPFNotification.Callback(){

    public void onServiceReady(SPFNotification spfNotification){
        List<SPFTrigger> triggers = spfNotification.listTriggers();
        ...
        spfNotification.disconnect();
    }

    public void onDisconnect(){...}

    public void onError(SPFError err){...}

});
```

A.7 SPFActivities

`SPFActivity` is an interoperable key-value data container that describes a potential or completed social action. As discussed in the next section it is designed to speed up the development of standard services and to ease the integration of different applications.

A.7.1 Data structure

Each activity features a set of standard fields:

Verb a string that identifies the type of the social action

SenderIdentifier and SenderDisplayName : details about the user who performed the activityÍ$\frac{3}{4}$

ReceiverIdentifier and ReceiverDisplayName : details about the user target of the activity.

The personal details of the sender and the receiver are automatically injected by the framework without the need of additional lookups; activities may also contain a map of fields needed to describe the action.

Activities can be used as parameters in methods of service interfaces, removing the need of additional parameters to pass context information like the name of the sender of a chat message.

Listing A.16: Using an activity for a chat.

```java
public void onMessageReceived(SPFActivity message) {
    ChatStorage storage = ChatDemoApp.get().getChatStorage();
    String senderId = message.get(SPFActivity.SENDER_IDENTIFIER);
    String senderName = message.get(SPFActivity.SENDER_DISPLAY_NAME);
    Conversation c = storage.findConversationWith(senderId);
    if (c == null) {
        c = new Conversation();
        c.setContactIdentifier(senderId);
        c.setContactDisplayName(senderName);
        storage.saveConversation(c);
    }

    Message m = new Message();
    m.setSenderId(c.getContactIdentifier());
    m.setText(message.get(ProximityService.MESSAGE_TEXT));
    m.setRead(false);
    storage.saveMessage(m, c);
}
```

A.7.2 Verbs routing

SPF implements an automatic routing service that dispatches activities to applications according to verbs. To access this functionality, you need to declare a service as a consumer of one or more activity verbs. Since more than one service can be consumer of a given verb, the user of SPF can select which of them is the default one by using the framework front-end. Sending activities to external applications, both local and remote, is possible through the method `sendActivity` of `SPFServiceExecutor` class. Listing A.17 and A.18 shows respectively the definition of a service and the invocation of a method based on the verb routing mechanism.

Listing A.17: Defining a verb consumer.

```
@ServiceInterface(
    app = "it.polimi.spf.demo.chat",
    name = "SPFChatDemo Proximity",
    description = "Service that allows users to communicate",
    version = "0.1",
    consumedVerbs = {
        ProximityService.POKE_VERB,
        ProximityService.MESSAGE_VERB }
)
public interface ProximityService {

    public static final String POKE_VERB = "poke";
    public static final String CHAT_VERB = "chat";

    @ActivityConsumer(verb = POKE_VERB)
    void onPokeReceived(SPFActivity poke);

    @ActivityConsumer(verb = CHAT_VERB)
    void onMessageReceived(SPFActivity message);
}
```

Listing A.18: Sending an activity.

```
SPFActivity message = new SPFActivity(ProximityService.CHAT_VERB);
```

```
message.put("text", text);
//spfPerson is an instance of SPFPerson retrieved from a search or a lookup
spfPerson.sendActivity(spfConnection, message);
```

Appendix B

Guide to SPF Framework

B.1 Introduction

This document is a guide to the APIs of *Social Proximity Framework* (*SPF*). The official repository of the project provides an example of spf provider app and different middleware implementations. `http://github.com/deib-polimi/SPF`

B.2 Overview

The framework is composed by two main components: `SPF`, which offers the access to all the functionalities provided by the framework, and an Android service, `SPFService`, which enables local applications to interact with these functionalities. The framework supports three kinds of actors: SPF-enabled applications installed on the same device (*local applications*), instances of the framework installed on other devices in proximity(*remote instances*) and the *SPF provider app* which is the topic of this guide.
Local applications can interact with the framework by means of SPFLib, a library that provides a set of endpoints to interact with the framework through IPC calls. On the other hand, remote instances can communicate with the local one through a proximity middleware which implements the transmission and reception of data through a communication channel. Two middlewares are provided with the framework: one based on Alljoyn, an open source middleware that enables communication among connected devices, and one based on Wi-Fi Direct, a Wi-Fi standard that enables devices to connect with each other with-

out requiring a wireless access point. The implementation of a new middleware will be described in section B.8.

B.2.1 Initialization

The `SPF` class is implemented as a sigleton, whose instance can be retrieved using `SPF.get()` method. However, before accessing an instance, the framework must be initialized, or an exception will be thrown. To initialize the framework, use the method `SPFContext.initialize`, passing as arguments an Android context that will be used in the framework, and a middleware to enable communication with remote instances. Listing B.1 shows how to initialize the framework.

Listing B.1: Initializing the framework

```

// Initializing with Alljoyn middleware
SPFContext.initialize(context, AlljoynProximityMiddleware.FACTORY);

// Initializing with Wi−Fi Direct middleware
// SPFContext.initialize(context, WFDMiddlewareAdapter.FACTORY);

// Now, we can obtain the SPF instance
SPF spf = SPF.get();
```

B.2.2 Configuring `SPFService`

The `SPFService` is active only when at least one local application is bound to it: as soon as no there are no more active connections, the service is terminated by the operative system. You can override this behaviour by starting the service in foreground. When in foreground, an Android service stays active in the background even if no application is bound to it. To start `SPFService` in foreground, you need to invoke the static method `startForeground` of the `SPFService`. Once the service is in foreground, it will stay active also when all the activities of your application are closed. To revert back to the default behaviour, use the static method `stopForeground`. Listing B.2 shows how to control the foreground status of `SPFService`.

Listing B.2: SPFService in foreground

```
// Start the foreground
```

```
SPFService.startForeground(context);

// Now the service is in foreground
// To revert back to the default behaviour
SPFService.stopForeground(context);
```

When a service is run in foreground, Android displays a notification to make the user aware of it. You can customize the notification displayed for `SPFService` by providing your `Notification` instance to `SPFContext` using the method `setServiceNotification`. Listing B.3 shows how to customize the notification shown for `SPFService`.

Listing B.3: Customizing SPFService notification

```
// Create a pensing intent to resume an activity
// when the notification is tapped
Intent intent = new Intent(context, MainActivity.class);
PendingIntent pIntent = PendingIntent
    .getActivity(context, 0, intent, PendingIntent.FLAG_UPDATE_CURRENT);

// Create the notification
Notification n = new Notification.Builder(context)
    .setSmallIcon(R.drawable.ic_launcher)
    .setTicker("spf is active")
    .setContentTitle("SPF")
    .setContentText("SPF is active.")
    .setContentIntent(pIntent)
    .build();

// Set the notification in SPFContext
SPFContext.get().setServiceNotification(n);
```

B.2.3 Listening for SPF events

`SPFContext` implements an event dispatching mechanism that is used by SPF components to dispatch events related to their functionalities. You can register your own listener by invoking the method `registerEventListener` providing your own implementation of the interface `SPFContext.OnEventListener`. The only method required by this interface is `onEvent(int eventCode, Bundle payload)`, where `eventCode` is the identifier of the event, while `payload` is a Bundle containing event-dependant data; the codes and the payload data will be described in the next sections. Once you're done listening for events, unregister the listener with `unregisterEventListener`. Listing B.4 shows how to initialize the framework.

Listing B.4: Registering an event listener

```
// Create a listener
SPFContext.OnEventListener listener = new SPFContext.OnEventListener() {
    public void onEvent(int eventCode, Bundle payload) {
        // Handle the event
    }
};

// Register the listener
SPFContext.get().registerEventListener(listener);

// Unregister the listener
SPFContext.get().unregisterEventListener(listener);
```

B.3 Security Monitor

The `SPFSecurityMonitor` controls the interaction between the local instance of SPF and external actors, which are SPF-enabled applications installed on the local device, and instances of SPF installed on remote devices. In particular, the monitor is divided into two components, the `ApplicationRegistry`, controlling the interaction with local applications, and the `PersonRegistry`, controlling those with remote instances.

B.3.1 Application registry

The `ApplicationRegistry` controls the interaction between the framework and SPF-enabled applications installed on the same device, by means of the interprocess communication capabilities offered by Android. Upon the first interaction, SPF requires an external application to register itself providing its `AppDescriptor`, which contains its label, package name and the list of permission, i.e. the list of SPF features it will utilize. Incoming requests are processed by a pluggable strategy that, by default, discards all incoming requests. This behaviour

can be customized by the SPF provider by implementing an `AppRegistrationHandler` and registering it in `SPFContext` with the method `setAppRegistrationHandler`.
The `AppRegistrationHandler` interface only requires the implementation of the method `handleRegistrationRequest`, whose parameters are:

- **context**: the Android context of the SPFService;
- **descriptor**: the `AppDescriptor` containing the details of the app that should be reviewed;
- **callback**: the callback to notify of the result of the review. In case of positive review, you should pass in the SPFPersona to be assigned to the accepted app.

In case the request is accepted, you should also provide the `SPFPersona` that the newly approved application will be allowed to access. Listing B.5 shows how to display a popup that lets the user review incoming requests.

Listing B.5: Handling app registration requests

```
SPFContext.setAppRegistrationHandler(new AppRegistrationHandler {

    public void handleRegistrationRequest(Context context,
        AppDescriptor descriptor, final Callback callback) {

        Dialog dialog = new AlertDialog.Builder(context)
            .setPositiveButton(android.R.string.yes,
                new AlertDialog.OnClickListener() {
                    public void onClick(DialogInterface dialog, int which) {
                        callback.onRequestAccepted(SPFPersona.DEFAULT);
                    }
            }).setNegativeButton(android.R.string.no,
                new AlertDialog.OnClickListener() {
                    public void onClick(DialogInterface dialog, int which) {
                        callback.onRequestRefused();
                    }
            }).create();

        // context is a service, thus to display a popup we need
        // the android.permission.SYSTEM_ALERT_WINDOW permission.
        dialog.getWindow()
            .setType(WindowManager.LayoutParams.TYPE_SYSTEM_ALERT);
        dialog.show();
    }
});
```

The list of authorized applications can be retrieved from `ApplicationRegistry` using the method `getAvailableApplications` which returns a collection of `AppAuth`, encapsulating the details of an application provided upon registration. The registry also allows to remove a previously allowed application using `unregisterApplication`, passing in the identifier of the app to remove. Listing B.6 shows how to manage the list of allowed applications.

Listing B.6: Managing the list of available applications

```
SPFApplicationRegistry registry = SPF.get().getApplicationRegistry();
List<AppAuth> apps = registry.getAvailableApplications();
for(AppAuth app : apps){
    String appName = app.getAppName();
    String appIdentifier = app.getAppIdentifier();
    Permission[] permissions = app.getPermissions();
    SPFPersona persona = app.getPersona();
}

// To remove an application
AppAuth app = registry.getAppAuthorizationByAppId("com.example.chat");
monitor.unregisterApplication(app);
```

B.3.2 Person registry

The `PersonRegistry` controls the interaction between the local SPF instance and remote one, ensuring that only allowed instances can access data stored in the profile. The clearance system works by dividing remote instances into *groups*, that are used to control the access to resources.

Sending a contact request

Before a remote instance can be added to a circle, you first need to add it as a *contact*. When an instance is added as a contact, its details, including the groups it belongs to, are saved in the `PersonRegistry`. Then, a contact request containing a unique token is sent to the remote

instance, where it is stored as a pending request: if the request is accepted, the details of the sender, including the token, are saved in the `PersonRegistry`, otherwise the token is discarded. When a SPF instance performs a request to a remote one, it includes the token shared by the two instances, if available, so that the recipient can verify the identity of the sender and provide him access only to the information he's allowed to access.
A contact request can be sent to a remote instance by invoking the method `sendContactRequestTo`, providing the following parameters:

- **targetUID**: the identifier of the instance the request should be sent to. Section B.4.1 shows how to search for remote instances and obtain their identifier;
- **passphrase**: a passphrase that will be used to encrypt the token. The recipient of the request will need to input the same passphrase to decrypt the token;
- **displayName**: The display name of the remote instance;
- **profilePic**: The picture of the remote instance;
- **groups**: The group the new contact should be placed in.

Upon reception, a request is saved in the `PersonRegistry`: the list of available requests can be obtained by means of the method `get-PendingRequests`, whose return type is a collection of `PersonInfo`, a container encapsulating the basic details of a remote instance including its identifier, display name and profile picture. A request can be confirmed using the method `confirmRequest`, providing the `PersonInfo` of the remote instance, the passphrase used by the sender to encrypt the token and the groups the new contact should be placed in; on the other hand, a request can be discarded with `deleteRequest`. Listing B.7 shows how to send, confirm and remove contact requests.

Listing B.7: Sending a contact request

```
PersonRegistry registry = SPF.get().getSecurityMonitor().getPersonRegistry();

// Sending a request
// person profile is the profile of the remote person
registry.sendContactRequestTo(
    personProfile.getFieldValue(ProfileField.IDENTIFIER),
    "my complex passphrase",
    personProfile.getFieldValue(ProfileField.DISPLAY_NAME),
    personProfile.getFieldValue(ProfileField.PHOTO),
    Arrays.asList("business"));

// Obtaining the list of requests
List<PersonInfo> requests = registry.getPendingRequests();

// Confirm a request
registry.confirmRequest(
    requests.get(0),
    "my complex passphrase",
    Arrays.asList("business"));

// Deleting a request
registry.deleteRequest(requests.get(0));
```

Upon reception of a contact request, an event is dispatched by SPF-Context: the event code is `SPFContext.EVENT_CONTACT_REQUEST_RECEIVED`, while the payload contains no data.

Managing contacts

The `PersonRegistry` allows you to retrieve the list of contacts already available using `getAvailableContacts`, which returns a collection of `PersonInfo`, and remove an existing contact using `deletePerson`, providing the `PersonInfo` of the contact to delete. Listing B.8 shows how to send, confirm and remove contact requests.

Listing B.8: Managing contacts

```
PersonRegistry registry = SPF.get().getSecurityMonitor().getPersonRegistry();

List<PersonInfo> contacts = registry.getAvailableContacts();
for (PersonInfo contact : contacts) {
    String identifier = contact.getIdentifier();
    String name = contact.getDisplayName();
}

registry.deletePerson(contacts.get(0));
```

Managing groups

Finally, the registry allows you to manage the list of groups and the assignment of instances. The method `getGroups` returns you the list of available groups, while `addGroup` and `removeGroup` allows you to add and remove a circle, respectively. To change the assignment of contacts to groups, use `addPersonToGroup` to add and`removePersonFromGroup` to remove a contact to/from a group.Listing B.9 shows how to send, confirm and remove contact requests.

Listing B.9: Managing groups

```
PersonRegistry registry = SPF.get().getSecurityMonitor().getPersonRegistry();

// Getting available groups
Collection<String> groups = registry.getGroups();

// Adding a group
registry.addGroup("Colleagues");

// Contact is a personInfo previously retrieved
registry.addPersonToGroup(contact, "Colleagues");
registry.removePersonFromGroup(contact, "Colleagues");

// Removing a group
registry.removeGroup("Colleagues");
```

B.4 Search

The SPFSearch API allows you to search for other instances of SPF in proximity of the user device, by specifying a description of the search to perform and a callback that will be notified when a search event happens. The search is performed by means of signals sent through the middleware to other instances of SPF; when a remote instances receives the signal, it replies back with a search result signal that includes as payload the basic information of the remote instance.
The description of a search includes three parameters: the number of search signals that will be sent to nearby devices, the time interval between two subsequent signals, and a search query, which is a collection of parameters matched against remote instances. A parameter can be of three types:

Predicate a profile field of the should have a specific value (e.g display name equals John Doe);

Tag specifies a word that should be contained in the profile;

Application specifies that the remote instances should have a specific app installed.

The callback, on the other hand, allows you to be notified of events related to your search so that you can react appropriately in your application. Events supported by the search manager are:

- **SearchStart**: when the search is started by the framework;
- **SearchStop**: when the search is stopped, either because the requested number of signal has been sent, or because the search request has been cancelled;
- **SearchError**: when an error occurs during the search;
- **SearchResultReceived**: when a new instance matching the query you provided has been found.
- **SearchResultLost**: when an instance previously found is no more available.

When you are notified of a search result the frameowrk provides you the unique identifier and the display name of the instance; the identifier can then be used with other components of

B.4.1 Performing a search

The `SPFSearchManager` allows you to start and stop searches by means of the two methods `startSearch` and `stopSearch`, respectively. The `startSearch` method requires as parameter the package name of your application, a `SPFSearchDescriptor` containing the configuration for the new search, and `SPFSearchCallback` to dispatch search events; the return value is the identifier for the newly started search, needed to forcedly stop it with `stopSearch`, and which is also passed as the first parameter in each method of the callback, to help you identify the search which triggered the event. Be careful that the methods of the callback are not called from the main thread, thus you can't interact with `View` objects directly.
Listing B.10 shows you how to search for people named John Doe.

Listing B.10: Searching for John Doe

```
SPFSearchManager manager = SPF.get().getSearchManager();
SPFSearchCallback.Stub callback = new SPFSearchCallback.Stub() {
    public void onSearchStart(String queryId) {
        // The framework has started the search
    }

    public void onSearchStop(String queryId) {
        // The search has ended
    }

    public void onSearchError(String queryId) {
        // there was an error during the search
    }

    public void onSearchResultReceived(String queryId, String userId,
        BaseInfo info) {
        // A John Doe was found
    }

    public void onSearchResultLost(String queryId, String userIdentifier) {
        // A John Doe is no more available
    }
};

int signals = 10, interval = 10*1000; // in milliseconds
SPFQuery query = new SPFQuery.Builder()
    .setProfileField(ProfileField.DISPLAY_NAME, "John Doe")
    .build();
SPFSearchDescriptor descriptor =
    new SPFSearchDescriptor(signals, interval, query);

// AppId is the package name of your app
String id = manager.startSearch("appId", descriptor, callback);

// In case you need to stop the search
manager.stopSearch(id);
```

B.4.2 Interacting with remote instances

Once you obtained the unique identifier of the instance you want to interact with, you can look up its remote stub in the `SPFPeopleM-`

`anager`, where SPF stores the remote stubs of instances detected in proximity. The connection to the remote instance is lazily initialized when the stub is used for the first time. Remote stubs are instances of the `SPFRemoteInstance` class which features methods to interact with the remote instance, as shown in the next sections. Listing B.11 shows you how to search for people named John Doe.

Listing B.11: Obtaining the stub for a remote instance

```
SPFPeopleManager manager = SPF.get().getPeopleManager();

//identifier is the unique id of a person found by means of a search.
SPFRemoteinstance instance = manager.getPerson(identifier);
if(instance == null){
    // The person is not available
}

// Interact with the instance
```

B.5 Profile

SPF offers `SPFProfileManager` to read and write information on a persistent user profile within a predefined set of fields. The support for `SPFPersona`, allows to store a different field value for each persona. Each profile field can also be assigned to a set of `groups` to control the access from remote instances of SPF.

B.5.1 Adding and removing personas

`SPFProfileManager` offers a simple yet complete API to manage the list of available `SPFPersona`. The method `getAvailablePersonas` allows you to obtain the list of available personas, while `addPersona` and `removePersona` respectively allows you to add and remove a persona, as shown in listing B.12.

Listing B.12: Adding and removing personas.

```
SPFProfileManager profile = SPF.get().getProfileManager();

// Adding a persona
profile.addPersona(new SPFPersona("Business"));

```

```
// Getting the list of available personas
List<SPFPersona> personas = profile.getAvailablePersonas();

//Removing a persona
SPFPersona persona = personas.get(0);
profile.removePersona(persona);
```

B.5.2 Reading and writing on the local profile

Each profile field is identified by an instance of the `ProfileField`, a class that encapsulates both the identifier and the type of the field. The `SPFProfileManager` supports only bulk operations to read and write multiple fields at the same time: You can obtain a list of values by calling `getProfileFieldBulk` passing in the array of ProfileField you want to read, and the persona from which the values should be read. The return value of this method is an instance of the `ProfileField-Container` class, offering a getter and a setter method to interact with the values: both these methods are generic depending on the type of the `ProfileField`. Listing B.13 shows how to retrieve the values of two fields from the profile.

Listing B.13: Reading from local profile.

```
SPFPersona persona = SPFPersona.DEFAULT;
SPFProfileManager profile = SPF.get().getProfileManager();
ProfileField<?>[] fields = {
    ProfileField.DISPLAY_NAME,
    ProfileField.BIRTHDAY
};
ProfileFieldContainer values = manager.getProfileFieldBulk(persona, fields);

String name = values.getFieldValue(ProfileField.DISPLAY_NAME);
Date birthday = values.getFieldValue(ProfileField.BIRTHDAY);
```

To modify the value of a field, simply call `setFieldValue` on the container for each value you want to change, then call the method `set-ProfileFieldBulk` to persist the changes. A container keeps track of the changes: to know if a container has been modified, use the method `isModified`. The status of a container does not reset automatically after the changes are saved, you have to do it by invoking `clearModi-`

`fied`; it is possible to reause the same container for more writes. Listing B.14 shows how to modify the value of the of a field.

Listing B.14: Writing to local profile

```
//values.isModified() returns false;
values.setFieldValue(ProfileField.DISPLAY__NAME, "John Doe");
values.setFieldValue(ProfileField.BIRTHDAY, new Date());
manager.getProfileFieldBulk(persona, fields);
//values.isModified() returns true;

manager.setProfileFieldBulk(values, SPFPersona.DEFAULT);
//values.isModified() returns true;

values.clearModified();
//values.isModified() returns false;
```

B.5.3 Assigning groups to fields

For each field, you can obtain the list of allowed groups using the method `getGroupsOf` and providing the `SPFPersona`, as the list is different for each persona; this method returns an Android Bundle with an entry for each `ProfileField`, where the identifier of the field is the key, while the list of groups is the value. To modify the list of circle, use the methods `addGroupToField` and `removeGroupFromField`, providing the field, the group name and the persona for which should be added. The procedure to get the list of all the groups created by the user was shown in section B.3.2, while listing B.15 shows how to manage a list of groups for a given field and a given persona.

Listing B.15: Groups management

```
SPFProfileManager manager = SPF.get().getProfileManager();
SPFPersona persona = SPFPersona.DEFAULT;

// Getting the groups of the field display name
Bundle groups = manager.getCirclesOf(mPersona);
List<String> nameGroups = groups.getStringArrayList(
ProfileField.DISPLAY_NAME.getIdentifier());

// Adding a group
manager.addCircleToField(ProfileField.DISPLAY_NAME,
    "My group", persona);
```

```

// Removing group
manager.removeCircleFromField(ProfileField.DISPLAY_NAME,
    "My group", persona);
```

B.5.4 Reading from remote profiles

In order to read profile data from a remote instance you need to obtain its remote stub as shown in section B.4.2. The remote stub has a method named `getProfileBulk` that behaves exaclty like its local counterpart: it will return a `ProfileFieldContainer` holding the values of requested fields, given that you have the permission to access them (see section B.3.2), otherwise the values will be null. Listing B.16 shows you how to read the field display name and birthday from a remote profile.

Listing B.16: Reading from a remote profile

```
ProfileField<?>[] fields = {
    ProfileField.DISPLAY_NAME,
    ProfileField.BIRTHDAY
};

// instance is the stub of the remote instance
// appId is the ID of your app
ProfileFieldContainer fields = instance.getProfileBulk(fields, "appID");
String name = values.getFieldValue(ProfileField.DISPLAY_NAME);
Date birthday = values.getFieldValue(ProfileField.BIRTHDAY);
```

B.6 Services

The Service API offered by SPF Framework allows you to retrieve the list of `SPFServices` and SPFActivity verbs supported by installed applications; it is also possible to set the default service to handle the SPFActivities of a given verb. The registration of a new service directly through the framework is not allowed.

B.6.1 Listing the SPFServices of an application

Before you can obtain the list of services registered by an application, you need to retrieve its packageIdentifier, as described in section B.3.1. Then, obtain a reference to `SPFServiceRegistry` and invoke its method `getServicesOfApp` passing as argument the packageIdentifier. Listing B.17 shows how to retrieve the name of each service registered by a given application.

Listing B.17: Obtaining the list of services of an app

```
SPFServiceRegistry registry = SPF.get().getSPFServiceRegistry();

// appId is the packageIdentifier of the target app
SPFServiceDescriptor[] services = registry.getServicesOfApp(appId);
for(SPFServiceDescriptor service : services){
    Log.d("Example","Service name: " + service.getServiceName);
}
```

The details of a SPFService are encapsulated in the class `SPFServiceDescriptor`, which offers the following getters:

- `getServiceName`: The name of the service;
- `getDescription`: A short description;
- `getAppIdentifier`: The identifier of the app which registered the service;
- `getVersion`: The version of the service;
- `getConsumedVerbs` : The SPFActivity verbs supported by the service.

B.6.2 Managing the routing of `SPFActivities`

The Service Registry allows you to obtain the set of verbs supported by all installed applications and, for each verb, the SPF service currently set as its default handler; this can be done by invoking the method `getSupportedVerbs` whose return type is a collection of the class `ActivityVerb`, offering the following methods:

- `getVerb`: the verb;

- `getSupportingServices`: the set of service supporting the verb
- `getDefaultService`: the service currently set as default for the verb.

To change the default service for a verb, use the method `setDefaultConsumerForVerb`, passing in the verb and the `ServiceIdentifier` of the new default consumer. Listing B.15 shows how to manage the list of supported verbs.

Listing B.18: Managing the list of supported verbs label

```
SPFServiceRegistry registry = SPF.get().getSPFServiceRegistry();
Collection<ActivityVerb> supportedVerbs = registry.getSupportedVerbs();
for(ActivityVerb supportedVerb : supportedVerbs){
    String verb = supportedVerb.getVerb();
    Set<ServiceIdentifier> consumers = supportedVerb.getSupportingServices();
    ServiceIdentifier def = supportedVerb.getDefaultService();
}

// Setting the first consumer as default
ActivityVerb first = supportedVerbs.get(0);
if(first != null && !first.getSupportingServices.isEmpty()){
    registry.setDefaultConsumerForVerb(
        first.getVerb(),
        first.getSupportingServices().get(0)
    );
}
```

B.7 Advertising and Notification

The SPF Advertising API allows a SPF instance to notify nearby instances of its presence by sending multicast signals, containing as payload a subset of the user profile, called "advertised profile". Using the `SPFAdvertisingManager` you can check whether advertising is active, turn it on/off, set which profile fields will be advertised and the persona from which their values will be read.
The Notification Manager, on the other hand, allows you to define `SPFTriggers`, reaction to incoming signals depending on the advertised profile. A `SPFTrigger` is composed of two parts: a `SPFQuery` that activates the trigger when a matching advertised profile is re-

ceived, and a `SPFAction` that defines the behaviour of the `SPFTrigger` once activated. Two types of actions are supported:

- **Intent**: SPF will send a broadcast intent locally on the device that received the advertising.
- **SendMessage**: SPF will send a notification, consisting of a title and a message, to the instance who broadcasted the advertising; received notifications are stored by the notification manager.

B.7.1 Advertising

Using the API provided by `SPFAdvertisingManager` you can check the state of advertising with `isAdvertising` and turn it on/off with `registerAdvertising` and `unregisterAdvertising` respectively.
The advertised profile can be set up with `addFieldToAdvertising` and `removeFieldFromAdvertising` to add/remove a field: both these methods accept a `ProfileField` as parameter. The identifier of the user is always included in the advertised profile.
You can set the `SPFPersona` used to advertise with `setPersonaToAdvertise` (section B.5.1 shows how to retrieve the list of available `SPFPersonas`); if you don't set a persona to advertise, the default one will be used. An example of how you can set up the advertising is provided in listing B.19. The framework can also advertise the list of installed applications that are linked to the current advertised persona. To manage this feature, use the method `setApplicationAdvertisingEnabled`. You can check whether the framework is currently set to advertise applications using the method `isAdvertisingApplications`

Listing B.19: Setting up advertisement

```
SPFAdvertisingManager manager = SPF.get().getAdvertisingManager();

// Add some profile fields to the advertised profile
manager.addFieldToAdvertising(ProfileField.DISPLAY_NAME);
manager.addFieldToAdvertising(ProfileField.BIRTHDAY);
manager.addFieldToAdvertising(ProfileField.INTERESTS);

// Set the persona to advertise
// persona is the reference to the SPFPersona
// that should be advertised
manager.setPersonaToAdvertise(persona);

```

```
// Start advertising
manager.registerAdvertising();

// From now on, SPF is avertising the profile
// To turn advertising off, use
manager.unregisterAdvertising();
```

When the advertising state changes, an event is dispatched by SPFContext: the event code is `SPFContext.EVENT_ADVERTISING_STATE_CHANGED`, while the payload contains a boolean indicating the new state, its key is `SPFContext.EXTRA_ACTIVE`.

B.7.2 Triggers

Triggers are managed by the Notification Manager, whose API allows you to list available triggers, as well as creating new ones and deleting existing ones. To create a new Trigger, you need to create a new instance of the `SPFTrigger` class. Its constructor requires the following parameters:

- **name**: a simple name for the trigger;
- **query**: a `SPFQuery` that will be matched against incoming advertised profiles. See section B.4.1 on how to create a new query;
- **action**: a `SPFAction` describing the behaviour of the trigger on incoming profiles in case the query matches. As said before, SPF provides two implementation, `SPFActionIntent`, which will dispatch an android intent with a given action on the local device, and `SPFActionSendNotification`, which will send a notification to the instance advertising its profile.
- **oneShot** and **sleepPeriod**: these two parameters describe the temporal behaviour of the trigger. If a triggger is oneshot it will fire only once XXX. For a non oneshot trigger, the sleepPeriod is the minimum interval of time between two consecutive activations.

Once a trigger object is created, you can save it in the `SPFNotificationManager` with the method `saveTrigger` which requires, besides the trigger instance, the package name of your app. The list of triggers saved in the notification manager by a specific application can be obtained with the method `listTriggers`; you can also delete all triggers registered by a specific application with `deleteAllTrigger`, or delete only one trigger with `deleteTrigger`. Listing B.20 shows how you can manage the list of installed triggers and create new ones.

Listing B.20: Managing triggers

```java
SPFNotificationManager manager = SPF.get().getNotificationManager();

// Get the list of triggered registered by the current app
List<SPFTrigger> triggers = manager.listTriggers("my.app.package.name");

// Create a new one-shot trigger
// that matches people named John Doe
// and sends them a message
SPFQuery query = new SPFQuery.Builder()
    .setProfileField(ProfileField.DISPLAY_NAME, "John Doe")
    .build();
SPFAction action
    = new SPFActionSendNotification("Hi", "Hi John from my app");
SPFTrigger trigger
    = new SPFTrigger("Greeting", query, action, true);
manager.saveTrigger(trigger, "my.app.package.name");


// To remove the newly created trigger
manager.deleteTrigger(trigger.getId(), "my.app.package.name");
```

B.7.3 Notification

Notifications received from other SPF instances are stored in the Notification Manager, and can be retrieved with the method `getAvailableNotifications`, whose return type is a collection of `NotificationMessage`; this class encapsulates all the details of the notification: the id of the sender, its title and its message. A single notification can be deleted using `deleteNotification` providing its ID, while you can remove all notifications using `deleteAllNotifications`. Listing B.21 shows how you can manage the list of installed triggers and create new ones.

Listing B.21: Managing notifications

```
SPFNotificationManager manager = SPF.get().getNotificationManager();

// Getting the number of notifications
int count = manager.getAvailableNotificationCount();

List<NotificationMessage> messages = manager.getAvailableNotifications();
for(NotificationMessage n : messages){
    String senderId = n.getSenderId();
    String title = n.getTitle();
    String message = n.getMessage();
    manager.deleteNotification(n.getId());
}

// To delete all notifications
manager.deleteAllNotifications();
```

Upon reception of a notification message, an event is dispatched by SPFContext: the event code is `SPFContext.EVENT_NOTIFICATION_MESSAGE_RECEIVED`, while the payload contains the received message with key `EXTRA_NOTIFICATION_MESSAGE`.

B.8 Implementation of a middleware

A proximity middleware is a communication system that enables the interaction between instances of SPF installed on different devices. A proximity middleware should offer two main functionalities:

- Detecting nearby instances on the same middleware and notifying them to the local SPF instance
- Supporting the interaction between two instances described in previous sections.

The implementation of a proximity middleware is composed by three interacting components:

- An implementation of the `ProximityMiddleware` interface, providing methods that enables SPF to control the behaviour of the middleware and to send multicast signals to nearby instances;
- An implementation of the `ProximityMiddleware.Factory` interface, that enables SPF to create a new instance of your middleware implementation when needed.

- A concrete subclass of the `SPFRemoteInstance` abstract class, providing methods to interact with a specific remote instance.

The factory class must implement the method `createMiddleware`, which should return a new instance of your `ProximityMiddleware`. The parameters provided you by SPF are:

- an **Android context** you can use in your middleware;
- an **InboundProximityInterface** to which you should dispatch incoming requests from remote instances
- the **identifier** of your SPF instance, corresponding to the IDENTIFIER profile field.

When your middleware detects a nearby instance of SPF, you should notify the local instance using the method `onRemoteInstanceFound` of the `InboundProximityInterface`; in the same way, you should notify when an instance is lost using `onRemoteInstanceLost`. Both these methods accept as parameter an instance of the `SPFRemoteInstance` interface, providing SPF the methods it needs to interact with the remote instances. These methods mirrors those of the `InboundProximityInterface`, and you middleware should act as a bridge between these endpoints: when a method of `SPFRemoteInstance` is invoked, you should serialize the parameters in a format compatible with your communication middleware, dispatch the invocation to the remote instance, deserialize the parameters and invoke the corresponding method on the `InboundProximityInterface`, then dispatching back the return value in the same way.

When creating a `ProximityMiddleware` instance, you should implement the following methods:

- `connect`, `disconnect` and `isConnected` are used to control the state of the middleware.
- `sendSearchSignal`: dispatches a search signal to instances in proximity.
- `sendSearchResult`: dispatches a search result signal to instances in proximity.
- `registerAdvertisement`, `unregisterAdvertisement` and `isAdvertising` are used to control the state of advertisement.

List of Figures

List of Tables

Listings

Bibliography

[1] WiFi-Alliance. Wi-fi peer-to-peer (p2p) technical specification v1.2, 2010.

[2] Daniel Camps-Mur, Andres Garcia-Saavedra, and Pablo Serrano. Device-to-device communications with wi-fi direct: overview and experimentation. *Wireless Communications, IEEE*, 20(3), 2013.

[3] IEEE 802.11u. Wireless lan medium access control (mac) and physical layer (phy) specifications: Amendment 9: Interworking with external networks. pages 1–208, Feb 2011.

[4] Android Developers. Wi-fi peer-to-peer. URL: `http://developer.android.com/guide/topics/connectivity/wifip2p.html` [cited Jan 2015].

[5] Allseen Alliance Inc. Alljoyn: A common language for internet of things. URL: `https://allseenalliance.org` [cited Jan 2015].

[6] Intel common connectivity framework. URL: `https://software.intel.com/en-us/ccf` [cited Jan 1015].

[7] Qualcomm Technologies Inc. Lte direct always-on device-to-device proximal discovery, August 2014.

[8] Open Mobile Alliance. White paper on mobile social network work item investigation. URL: `http://technical.openmobilealliance.org/document/OMA-WP-Mobile_Social_Network-20110516-A.pdf` [cited Jan 2015].

[9] Json activity streams 2.0. URL: `https://tools.ietf.org/html/draft-snell-activitystreams-09` [cited Jan 2015].

[10] OpenSocial Specification 2.5.1. URL: `http://opensocial.github.io/spec/2.5.1/OpenSocial-Specification.xml` [cited Jan 2015].

[11] Matthias Häsel. Opensocial: An enabler for social applications on the web. *Commun. ACM*, 54(1):139–144, January 2011.

[12] OpenSocial Social Gadget Specification 2.5.1, August 2013. URL: `http://opensocial.github.io/spec/2.5.1/Social-Gadget.xml` [cited Jan 2015].

[13] OpenSocial Social Server API Specification 2.5.1, August 2013. URL: `http://opensocial.github.io/spec/2.5.1/Social-API-Server.xml` [cited Jan 2015].

[14] OpenSocial Social Data Specification 2.5.1, August 2013. URL: `http://opensocial.github.io/spec/2.5.1/Social-Data.xml` [cited Jan 2015].

[15] Open Mobile Alliance. Social Network Web Enabler, August 2013. URL: `http://technical.openmobilealliance.org/Technical/technical-information/release-program/current-releases/snew-v1-0` [cited Jan 2015].

[16] Ahmed Karam and Nader Mohamed. Middleware for mobile social networks: A survey. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 1482–1490. IEEE, 2012.

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

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

[19] Steffen Kern, Peter Braun, and Wilhelm Rossak. Mobisoft: an agent-based middleware for social-mobile applications. In *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, pages 984–993. Springer, 2006.

[20] Dimitris N Kalofonos, Zoe Antoniou, Franklin D Reynolds, Max Van-Kleek, Jacob Strauss, and Paul Wisner. Mynet: A platform for secure p2p personal and social networking services. In *Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on*, pages 135–146. IEEE, 2008.

[21] Dario Bottazzi, Rebecca Montanari, and Alessandra Toninelli. Context-aware middleware for anytime, anywhere social networks. *Intelligent Systems, IEEE*, 22(5):23–32, 2007.

[22] Alessandra Toninelli, Animesh Pathak, and Valérie Issarny. Yarta: a middleware for managing mobile social ecosystems. In *Advances in Grid and Pervasive Computing*, pages 209–220. Springer, 2011.