SPF: A Middleware for Social Interaction in Mobile Proximity Environments

Luciano Baresi*, Laurent-Walter Goix§, Sam Guinea*,
Valerio Panzica La Manna‡, Jacopo Aliprandi*, and Dario Archetti*
* Dipartimento di Elettronica Informazione e Bioingegneria
Politecnico di Milano,
via Ponzio 34/5, 20133 Milan, Italy
Email: {luciano.baresi|sam.guinea}@polimi.it
{jacopo.aliprandi|dario.archetti}@mail.polimi.it
§ Econocom-Osiatis,
75 cours Albert Thomas, 69003 Lyon, France
Email: laurent.goix@econocom-osiatis.com
‡ MIT Media Lab, 20 Ames St.,
02139 Cambridge, MA, USA
Email: vpanzica@mit.edu

***Abstract*—Smart interconnected devices are changing our lives and are turning conventional spaces into smart ones. Physical proximity, a key enabler of social interactions in the old days is not exploited by smart solutions, where the social dimension is always managed through the Internet. This paper aims to blend the two forces and proposes the idea of *social smart space*, where modern technologies can help regain and renew social interactions, and where proximity is seen as enabler for dedicated and customized functionality provided by users to users. A Social Proximity Framework (SPF) provides the basis for the creation of this new flavor of smart spaces. Two different versions of the SPF, based on different communication infrastructures, help explain the characteristics of the different components, and show how the SPF can benefit from emerging connection-less communication protocols. A first assessment of the two implementations concludes the paper.**

I. Introduction

Smart devices are rapidly and deeply changing our lives. From smart-phones to televisions, washing machines, coffee machines, refridgerators, watches, and glasses, communication is possible at all times and through different appliances. A key enabler of this revolution has been the availability of communication protocols: from conventional Wi-Fi and 2G/3G/4G networks to new emerging, connection-less protocols, such as WiFi-Direct [1] and LTE-Direct [2].

Even if physical spaces are becoming *smart spaces*, where people can seamlessly interact —mainly through their mobile devices— with appliances and other people in the same area, social proximity has often been neglected and seen as accidental. For example, commuters still tend to chat with their friends through well-known cloud-based social networks, and do not exploit physical proximity to socialize with new people.

This has slowly been changing, and a new breed of social applications that have physical proximity as a cornerstone has started to emerge. FireChat[1] is a mobile application that allows you to chat with everyone around you, even when there is no Internet connection available. Bizzabo[2] is an event management platform that allows event organizers to maximize attendee engagement with one-on-one mobile and proximal messaging. Another experiment in this direction is TrainRoulette [3]. It studies the impact that social mobility has on the travel experience, with a focus on anonymous social interactions between nearby commuters.

Both industry and academia have proposed interoperable communication layers and middleware infrastructures that could be used to create these kinds of applications. For example, solutions like AllJoyn [4] and Intel CCF [5] can be used to create smart spaces by letting participants share, discover, and exploit available functionality (services). However, the focus has been —until now— on managing the heterogeneity of communication protocols and operation systems, but not on *social proximity*. These approaches still do not see social proximity as an added value, and thus as an enabler for dedicated services; it is neglected and replaced by the idea that one can always be connected to everybody.

This paper takes a new viewpoint, one that wants to exploit social proximity as an added value. We advocate that modern spaces should not only be *smart*, but *social* as well. The social dimension matters, and it should be used to exploit special-purpose services, or to customize existing ones to special needs and contexts. We believe that user-centric interactions are key to blending new technologies with the identities of the different individuals. As a result, each individual can exploit technology-mediated interactions to show different identities and exploit different services.

The idea of a *social smart space* is fostered by the creation of a **Social Proximity Framework** (SPF)[3], as a means and mediator to support and ease the creation of smart spaces

Laurent-Walter Goix and Valerio Panzica La Manna carried out the work reported here when they were with Telecom Italia, Italy, and Politecnico di Milano, Italy, respectively.

[1]https://opengarden.com/apps
[2]https://www.bizzabo.com
[3]The Social Proximity Framework (SPF) is an open source project available at https://github.com/deib-polimi/SPF.

with a social flavor, and where physical proximity matters. The SPF allows people, spaces, and appliances to be dynamically discovered in proximity, on the basis of revealed information that can be customized according to the context and to privacy settings. The SPF is currently available for Android devices, in two different implementations: one based on the AllJoyn middleware [4] and one implemented directly on top of WiFi-Direct [1]. The two implementations were chosen so that we can understand and explain how social smart spaces can benefit from novel connection-less communication protocols.

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 or apps[4] 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 rest of the paper is structured as follows. Section II exemplifies the idea of social smart space and defines the requirements for a social proximity framework. Section III describes the general architecture of the SPF. Section IV describes the two versions of the SPF we have implemented. Section V proposes a first assessment of the framework. Section VI discusses related work, and Section VII concludes the paper.

II. Requirements for a Social Proximity Framework

From a business standpoint, various discussions with potential customers and partners allowed us to identify concrete application scenarios that would benefit from a common framework able to manage social interactions and identity in proximity. These refer mainly to owners of small, local shops, restaurants, or bars and to mobility spaces managed by transport operators.

Social Networking in Proximity (SNiP) is a generic use case that allows the users in a physical space to interact with each other on the basis of common interests, or goals. The SNiP service can be customized for different domains and contexts. On a weekday professionals can advertise their skills, to receive offers related to open positions. At fairs and large exhibitions, customers who are interested in particular products or businesses can be discovered by related companies and managers. During a party or a leisure trip users can discover and meet new people on the basis of common recreational interests or can self-organize (e.g. for ride sharing planning, collaborative problem-solving related to a disruption or an emergency, etc).

A SNiP profile usually consists of a username, a picture, and a set of arbitrary/predefined categories that represent skills, interests, or goals. Such data can be imported —and maybe integrated— from traditional, cloud-based social networks, or remain local. In a purely proximity-based context, users can advertise the profiles stored on their devices, and then be discovered by the others in the surrounding space. There would be no involvement of cloud-based technologies, neither would there be any GPS-based proximity calculation —that are not available in several contexts such as on transportation means, in indoor spaces, or when roaming. The owners of the smart space may also decide not to provide any WiFi connectivity to their customers, and thus any (social) interaction should be routed through local, ad-hoc solutions. Furthermore to preserve their privacy and security, users may decide to only advertise parts of their profiles, or to manage multiple views (*personas*) on them based on their mood or context. Once users discover interesting profiles, they can connect to each other and exchange *activities* such as messages, images, and other media content, directly.

Taking a more retail-centric approach, the owners of shops, bars, and restaurants may be keen on tailored and local advertisements on the basis of the preferences of their customers. As an example, users should be able to exploit the framework to declare their interest in receiving coupons and offers related to "Belgian Beers". Shops and pubs selling Belgian beers can use the framework to discover customers interested in their products in proximity, and perform targeted marketing campaigns. They may provide coupons, which become available only when a user enters the shop, or provide additional services and discounts for frequent users. When the advertisement is issued, the framework should perform multiple, and possibly periodic, searches for profiles that match the advertisement. The matching is carried out by comparing user-provided profiles and the business goal of the space owner (i.e., the profile of the user associated with the commercial activity). Every matching profile (client) then receives a push notification with the targeted advertisement.

The idea of proximity-based marketing campaigns is not new. As an example, recent solutions have involved the use of iBeacons or other indoor positioning systems, installed in malls and shops, to discover the presence of the customers within the store. However, such services do not natively support social identities, profiles and exchange of activities. They also traditionally require ad-hoc cloud-based solutions to generate personalized advertisements. A sound, smart framework should simplify these activities by adding the social layer to the smart space.

A. Requirements for the SPF

On the basis of the scenarios described above we now summarize the main requirements a novel user- and context-aware framework needs to satisfy to support social interaction in proximity.

Profile Management: The framework must allow for the creation and management of a social profile stored on the user's device. A profile contains a user identifier, a picture, and a set of interests. As privacy options a user identifier can be: (i) *anonymous* and not discoverable in public; (ii) *public*, e.g., a name that is imported from a traditional cloud-based public social profile; (iii) *opaque*, which means that it can be mapped to a real identity by acquaintances, for example, but it cannot be recognized by unknown users.

[4]The words *service* and *app(lication)* are often used as synonyms throughout the paper.

Search: the framework needs to support profile discovery (search) in proximity. The discovery should be based on querying the profiles that are in proximity, and it should provide a mechanism for filtering profiles against matched interests. Discovery can be fully peer-to-peer amongst users or it may be mediated by a centralized device installed on the smart space (for example in case of a shop owner).

Advertisement: profile advertisement should be performed in different ways based on privacy settings and context. A profile can be partially or totally advertised or provide different information on the basis of the context or of the specific application used. Advertisements can also be associated with specific services, or specific group of users.

Services: The framework should support extra customizable services related to social interaction, which could be made dynamically available to all the applications relying on the framework.

Activities: Similarly to traditional social networks, the framework should support different types of common activities such as textual posts, media content, and push notifications.

Security: The framework should provide a means to control how the information that is saved in a profile is shared with applications, and also a means to prevent unauthorized access to specific features of the framework.

The framework should also provide high-level and extensible communication facilities to support the interactions among users in proximity. The communication facility should mask the underlying communication protocols adopted for supporting the interaction and the devices associated with the profile. Specifically, it should support both ad-hoc proximity and infrastructure-based networks to better fit the technical and business constraints of the (owner of the) smart space.

These requirements represent open research questions. The solutions could be borrowed from existing systems, but they should be tailored to the context of proximity-based communication and mobile platforms. The SPF we propose addresses these issues.

III. Social Proximity Framework

Our Social Proximity Framework (SPF) aims to ease the development of applications that want to exploit social interactions as key enablers for dedicated data exchanges and service provisioning.

Figure 1 shows a high level overview of the framework, as deployed onto two connected devices. The SPF consists of two layers: the SPFCoreLayer and an underlying CommunicationLayer. The former is responsible for mediating all social interactions, while the latter is responsible for providing inter-device connectivity. The SPF was architected to make it pluggable with respect to different communication layers. In fact, Section IV discusses two diverse implementations, one based on the AllJoyn middleware [4] and one developed on top of the Android implementation of WiFi Direct [1].

To ease its distribution, the SPF is packaged into a single Android application called SPFCoreApp. This application contains the actual framework, and an accompanying default graphical user interface. This interface allows one to configure the various aspects of the framework, such as the user's social profile, rules for accessing local and/or remote profiles, etc.

Applications that reside on the device, and that want to make use of the capabilities offered by the SPF, can bypass the UI and interact directly with the SPFCoreLayer. This layer, in fact, exposes a programming interface for inter-app communication that has been defined using AIDL (i.e., the Android Interface Definition Language). To simplify the development of SPF-enabled apps, we also provide an SPFLibrary, which can be included directly into any app. It offers all the capabilities provided by the SPF, and hides the intricacies of the inter-app communication that occurs between the app being built and the SPFCoreLayer.

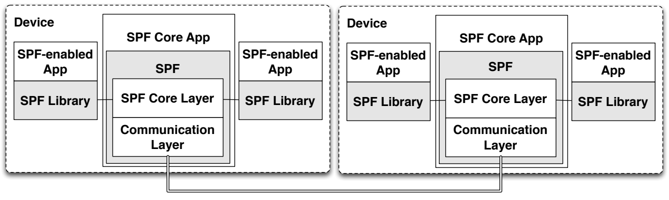


Fig. 1. SPF-enabled App Interactions.

Figure 2 provides a more in-depth representation of the architecture of the SPF on a single device. The SPFCoreApp provides two interfaces. We have already discussed `LocalAppInterface` for inter-app communication. `RemoteSPFInterface`, on the other hand, is used to connect the SPFCoreLayer to the underlying communication middleware. It makes use of a generic `MiddlewareInterface`, which must then be properly implemented by the underlying middleware directly, or through a series of `MiddlewareAdapters`.

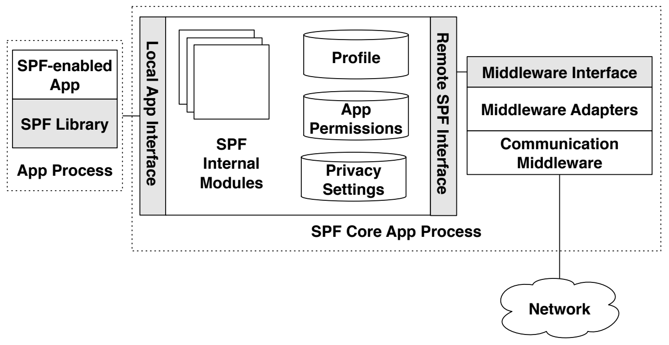


Fig. 2. Architecture of the SPF.

Within the SPFCoreLayer we have the software modules that provide the six main capabilities of SPF: *Profile Management*, *Search*, *Service Registration*, *Advertisement*, *Activities*, and *Security*. We also have data storage, where we store profiles, application permissions, and privacy settings.

A. Profile Management

The SPFCoreApp allows users to store personal profiles, whose information model is inspired by *OpenSocial*[5], that cover both personal information and additional social data.

[5]http://opensocial.org

The data are stored as a simple collection of fields; they can be modified directly by the user through the UI, or by local or remote applications according to specific privacy settings, which we will discuss in more detail in the upcoming section on security.

The SPF introduces a means to diversify the information that is provided to each application. This diversification goes under the name of *personas*. A persona is a named collection of profile fields; the main novelty is that the SPF allows different personas to assign different values to the same fields. As an example, the *interests* profile field describing user interests may have different values in the different personas.

The same person (device) can exploit different personas according to his/her needs. Since each application is associated with a specific persona, one can create, for example, a business-oriented persona, and differentiate it from one that is used for more "personal" applications. The creation, deletion, and management of the personas is achieved through the SPFCoreApp.

B. Search

Search is the mechanism that applications use to discover remote SPF instances. Searches are performed through the exchange of broadcast messages that leverage the communication primitives provided by the underlying communication middleware, as we will discuss in Section IV.

Searches consist of simple queries that predicate over specific profile fields. Empty queries return all the SPF instances that are in the proximity of the query issuer. If one wants to limit the query to a specific persona, this can be achieved by having the query contain the particular identifier of that persona. Search queries also require the specification of a *search duration* and a *search frequency*. This way the query can be periodically re-iterated for the entire specified duration. This is important since the search results will change as the space and location of the particular device change. Moreover, SPF supports multiple concurrent searches, and guarantees real-time updates to the search results.

C. Advertisement

Advertisement is the feature that allows users to advertise their profiles. Advertising is tied to the life cycle of the SPFCoreApp, meaning that it is possible to repeatedly broadcast specific profile personas as long as the SPFCoreApp network resources are active.

Advertisement can be useful in a scenario involving targeted marketing campaigns (as described in Section II). In this scenario, profile information, advertised by the user's app, can be captured by a store's SPF-enabled service, which can then proceed to send targeted coupons/offers back to the user.

To react to the reception of an *SPFAdvertisement*, the SPF introduces the notion of *SPFTrigger*. An *SPFTrigger* consists of an event/condition/action (ECA) rule. Events and conditions are expressed as *SPFSearch* queries, while actions can be triggered both locally and remotely, and can be of two types: *SPFActionIntent* and *SPFActionSendNotification*. In the former case, an actual Android Intent is broadcasted and received by the device that registered the trigger, while, in the latter case, a simple textual notification is shown to the recipient.

An *SPF trigger* can also define a *sleep period*, that is an interval in milliseconds within which no other actions should be performed, and whether the action should only be executed once in the SPF-enabled app's life cycle (i.e., whether the action is *one shot*).

In other words, the advertising feature allows for the definition of "persistent queries", which trigger actions whenever a match is found. This makes development much easier, since there is no need for the application developer to deal with background *SPFSearches*. Leaving this task to a shared SPF instance allows for a more optimized use of the resources.

D. Services

SPF, however, is not just about sharing profile information. It is also about enabling social interactions. This means that SPF-enabled applications are allowed to invoke functionality (services) provided both locally and by other devices. This is achieved through the notion of *SPFService*.

An *SPFService* is defined by a Java interface specified through a dedicated annotation (`@SPFServiceInterface`), and contains various kinds of meta-data, such as the package identifier of the app that has registered the service, its name, and its version.

Service registration starts when an SPF-enabled app makes a call to its local SPFLibrary to register a new service. This kicks off a two-step process: a validation step, which is performed locally by the SPFLibrary itself, and an actual registration step, which is performed by the SPFCoreApp running on the device. The validation step checks the interface being registered to see if its methods use valid input and return parameters. The registration step adds the service to a lookup table that is then used during searches. Services are identified by the unique Google Play Store identifier of the app and their name.

To dispatch incoming requests to an *SPFService*, the SPFCoreApp needs to be able to connect to the application that registered the service, even if that application is not running at that time. This is achieved through Android Services, a solution that allows application components to perform background operations. This is why one must also provide a suitable Android Service when registering an *SPFService*.

The invocation of an *SPFService* is made through the SPFCoreApp, which performs a local *SPFService* lookup. When the service is found, the SPFCoreApp binds to the third-party app that exposes the service, and the actual service execution can take place. How remote binding is achieved depends on the nature of the underlying communication framework. We refer the reader to Section IV, where this is explained in detail. Finally, access to *SPFService*s is regulated by specific access rules. This will be discussed in the upcoming section on security.

E. Activities

Activities are intended as means to simplify the development and integration of typical well-known social functionality. *SPFActivities* are inspired by *Activity Streams*[6]. In this specification, activities are used as a semantic description of potential or completed actions. Concrete examples of Activities are, therefore, the posting of a message on a timeline, the posting of a picture on a chat, etc.

SPFActivities have a standard set of data that are automatically injected by the SPF. They have information about the sender and the receiver, the date and time at which the action is performed, and (possibly) the location where it occurs. An *SPFActivity* also contains a "verb", that is, a string that identifies the type of social action that is being performed. Examples of verbs are *post* and *share* that correspond to the common functionality of posting and sharing content in traditional social networks.

SPFActivities are used in two different ways within the SPF: as a means to enrich the parameters of an *SPFService* method, and as a means to provide "lower-level integration". This latter case means being able to interact with an application without knowing about its service interface. This is supported by an automatic verb-routing mechanism, which dispatches requests based on the verb used in the *SPFActivity*. The verb-routing mechanism provides a means to support interoperable and application-agnostic social interactions. For example, an *SPFActivity* with verb *post* sent by an application can be received by a different application. To enable this kind of interaction, a developer must enrich the *SPFService*'s definition with annotations that state that it can also be considered an *SPFActivity* consumer, for certain verbs.

F. Security

Profile management at the application level, which is what happens with personas, is not always enough. Sometimes the way the information is socially shared depends on the level of familiarity the user has with another person, regardless of the app being used. This is why, on top of personas, the SPF provides a "circles" based clearance system, similar to the one provided by Google+.

The clearance system allows SPF users to categorize the people they interact with into circles, and then limit the access to specific profile fields only to some circles. This is achieved through a symmetric token mechanism. When two SPF users, let us say users A and B, come into proximity of one another, one user (e.g., A) sends a contact creation request to the other (e.g., B). Before the request is actually sent, user A selects the circles user B will be placed in, using the SPFCoreApp. This results in the creation of a unique token for users A and B that is stored locally within user A's SPF instance. This token is then sent as a part of the contact creation request. Upon reception, a notification in user B's SPFCoreApp tells the user that s/he must review the request, and confirm or deny it. If user B accepts the request, s/he is then asked to identify the circle(s) within which user A should be placed. Finally, the token is stored locally in user B's SPF instance as well. After a successful contact creation process, both parties hold the shared symmetric token. At this point, whenever an SPF instance contacts another SPF instance to request some profile information, the token must be passed as a part of the request. This allows the recipient to verify whether that specific person actually has access to the profile information.

How SPF-enabled applications access SPF capabilities, on the other hand, is managed in a way that is similar to how OAuth works in common Web applications. The first time an SPF-enabled application is run on a device, a graphical user interface is presented to the user. This UI declares what SPF capabilities the SPF-enabled app would like to access, and asks the user for explicit permission.

To avoid granting access to malicious apps that may attempt to connect to the SPF, the app must present its "unique" Google Play Store identifier. Only if the credentials are valid will the access request be allowed to proceed, and will the graphical UI actually be presented to the user.

The SPF capabilities that an app can request access to are: (a) registration of SPF services for other applications, (b) initiation of a proximity-based search, (c) execution of services published locally by other apps, (d) execution of services published remotely by other apps, (e) reading of a local profile, (f) reading of a remote profile, (g) writing on a local profile, and (h) access and use of the SPF Advertisement facilities.

As soon as access is granted, a token is generated for the requesting application. From this point on, all the requests to the SPF made by that app must include this token to let the SPF check whether it has been granted access to the capabilities it is attempting to use. Furthermore, the SPFCoreApp contains a list of all the permissions that have been granted to the other registered apps, so that the user can modify them according to his/her needs.

IV. Implementation

As we have seen, the SPF provides developers with many different tools that make the development of social proximity apps much easier. Tools like profile management, search, and service registration are possible thanks to an underlying communication middleware that deals with, and hides, the complex intricacies of device connectivity.

We designed the SPF to ease the adoption of different underlying communication infrastructures. This section discusses the two implementations we have created. The first is based on AllJoyn [4], while the second is built directly on top of the Android implementation of WiFi Direct [1]. While AllJoyn currently requires the presence of a fixed WiFi Access Point, WiFi Direct was chosen as a way to test the SPF in the absence of a networking infrastructure. A first detailed comparison of these two implementations is presented in Section V.

A. AllJoyn Implementation

AllJoyn is an open source project, initiated by Qualcomm and currently managed by the AllSeen Alliance, which provides a software framework for enabling interoperability among various types of devices. Since it focuses on scenarios such as connected homes and automotive, participating devices can go from smart-phones and tablets to low-level appliances and media equipment.

AllJoyn is cross-platform, cross-brand, and supports different kinds of connectivity means (e.g., Bluetooth, WiFi, etc.).

[6]http://www.w3.org/TR/activitystreams-core/

It is not a low-level communication protocol, but a set of services that allow developers to focus on higher-level problems. Some of the services it provides are discovery, capability broadcasting, remote procedure call, interface sharing, etc. These features made it a strong candidate for implementing the underlying communication layers of the SPF, and was our first implementation choice.

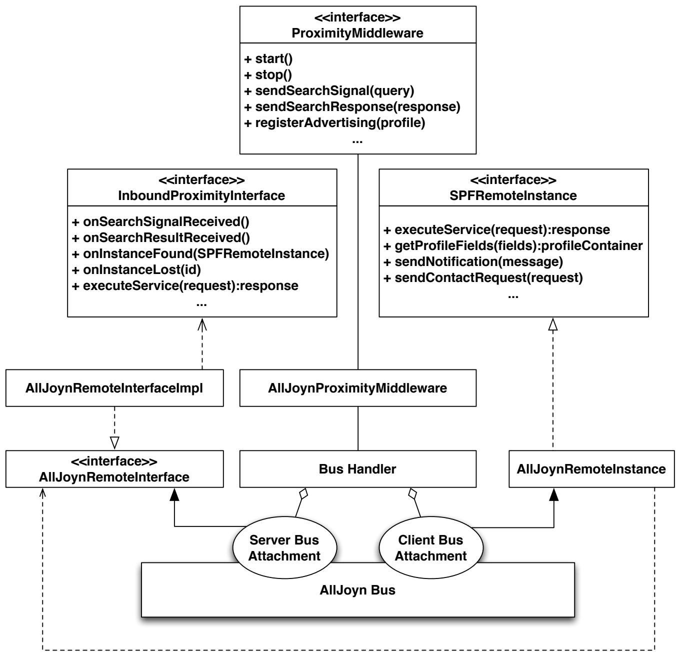

<<interface>>
ProximityMiddleware
+ start()
+ stop()
+ sendSearchSignal(query)
+ sendSearchResponse(response)
+ registerAdvertising(profile)
...
<<interface>>
InboundProximityInterface
+ onSearchSignalReceived()
+ onSearchResultReceived()
+ onInstanceFound(SPFRemoteInstance)
+ onInstanceLost(id)
+ executeService(request):response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

Fig. 3. Plugging AllJoyn into the SPF.

Figure 3 shows how AllJoyn implements the three framework-agnostic interfaces that make the SPF pluggable with respect to the underlying communication middleware: `ProximityMiddleware`, `InboundProximityInterface`, and `SPFRemoteInstance`. The first interface is used to manage the life cycle of the connection that is made to the underlying communication middleware. It is also used to initiate SPF capabilities that require a connection, such as searches and advertisements. The second interface is used to manage requests that are received from the underlying communication middleware. In particular, it provides methods that allow the SPF to respond to search requests, advertisement signals, notifications about SPF instances being found or lost, etc. It also manages incoming service execution requests. The third and last interface is used to manage outgoing connections to other SPF instances. In particular, it allows the SPF to make calls to remote SPF services, to request profile information, and to send contact creation requests.

The AllJoyn system consists of a virtual bus that connects distributed AllJoyn daemons that are installed onto the participating devices. Every application that connects to the virtual bus does so via a Bus Attachment. In our SPF implementation the class that directly interacts with the AllJoyn bus is called `BusHandler`. It is an Android Handler that runs in its own Looper thread, and provides asynchronous communication between AllJoyn and the upper layers of our implementation. The Bus Handler connects to the device's AllJoyn daemon through two AllJoyn Bus Attachments: `ClientBusAttachment` and `ServerBusAttachment`. The first is used to interact with remote SPF instances, while the latter for managing incoming requests.

The complete details of how AllJoyn is used to implement all of the SPF's capabilities would be too much for this paper. We will, however, focus briefly on how discovery is achieved in AllJoyn, and on how we used it to implement the search functionality. The reason for concentrating on discovery and search is that it will also allow us to highlight the key problems and differences that arose when we developed our WiFi Direct based implementation.

AllJoyn provides a discovery feature that is based on the fact that Bus Attachments can advertise software components on the network using unique identifiers (e.g., `it.polimi.spf.<name>`). Other Bus Attachments, on the other hand, can express interest in the arrival or departure of components that satisfy a given prefix (e.g., `it.polimi.spf`). This discovery mechanism is always active. This means it continuously produces arrival and departure event messages that are managed by methods `onInstanceFound` and `onInstanceLost` of class `InboundProximityInterface`. The execution of these methods leads to the creation or destruction of `AllJoynRemoteInstance` objects. These are the actual proxy objects that are used to achieve remote SPF interactions. They are lazily initialized. At the beginning they are empty objects that only contain the remote instances' identifiers; however, when the framework needs to initiate an actual exchange, an appropriate AllJoyn session is created.

On top of this discovery mechanism we have our search mechanism. Search is the application-level capability that third party apps use to discover other SPF instances. It is implemented using AllJoyn signals, i.e., unreliable broadcast messages that are sent over the virtual bus. These signals are captured and managed by the SPF instances that are on the bus, through method `onSearchSignal` of class `InboundProximityInterface`. Every remote instance that satisfies the query issues a search result. These results are received by the issuer of the search, through method `onSearchResult` of class `InboundProximityInterface`. The search result contains the remote application's identifier, which allows the SPF to find the corresponding `AllJoynRemoteInstance` object, created through the AllJoyn discovery mechanism, that it should use from that point on to achieve the remote interactions.

Since AllJoyn discovery is always on, its results are continuously fed into SPF Search, allowing the search mechanism to be continuously up-to-date with respect to the actual situation seen on the virtual bus.

B. WiFi Direct Implementation

Using the same three high-level SPF interfaces, we set out to implement a second version of the SPF that substitutes AllJoyn with WiFi Direct. WiFi Direct offers a lower-level network abstraction with respect to AllJoyn. It is a standard that allows enabled devices to connect and communicate using WiFi speed, even in the absence of a central access point.

AllJoyn requires an existing IP local network and as such does not fully support Wifi-Direct unless a connection has been already established.

WiFi Direct works by creating groups of connected devices. When a group is created, a single device is elected to be the *group owner*. This means that it is responsible for creating a Soft(-ware) Access Point to which all the other components may connect. Although it provides some high-level services such as the discovery of nearby devices, the actual communication is still based on standard socket technology.

Unfortunately we soon discovered that WiFi Direct presented some constraints that did not make it a good candidate for implementing the SPF. In particular, it made it hard to replicate the collaboration between discovery and search that was so successful in the AllJoyn implementation.

A first constraint that the Android implementation of WiFi Direct has is that, in general, peer discovery is extremely slow, taking up to 60 seconds in some cases. This is due to a constant defined at the operating system level that limits the updates of the list of available peers. A second, and even more important constraint, is that it is impossible with WiFi Direct's discovery API to update the information being used to perform service matching. When a WiFi Direct enabled device wants to register a service for discovery, it needs to provide a HashMap containing information about that specific service. The Android implementation of WiFi Direct then uses callback methods to notify an application of an available service. Unfortunately, this HashMap cannot be updated without re-registering the service, and cannot therefore be used as a means to spread the query signals like we did in the AllJoyn implementation.

For these reasons we decided to build an additional intermediate middleware layer, to be placed between WiFi Direct and the SPF, called *WFD_2_SPF*. The goal was to make WiFi Direct more suitable for our uses.

Our solution is based on the construction of an overlay network driven by WiFi Direct's notion of groups. We build a star-topology overlay whose root node is a WiFi Direct group owner. All other group members are connected solely and directly to the group owner through a socket connection. This allows us to improve peer discovery by having the group owner dispatch instance discovery messages whenever a new SPF instance enters or leaves the network: this is easy for the owner to detect. It also allows us to improve general messaging, since the group owner can simply route the messages to their intended destinations.

WFD_2_SPF was also used to provide higher-level communication abstractions for the SPF. In particular, we introduced three communication primitives: one for sending broadcast messages, one for sending unicast messages, and an RPC-friendly primitive for sending a message and waiting for its response.

Figure 4 illustrates the architecture of WFD_2_SPF. At the core of our implementation lies class `GroupActor`. This is an abstract class that contains the logic that is common to group owners and regular group members. In particular, it contains methods to connect and disconnect, to perform messaging, and to respond to incoming messages. This class is extended by `GroupClientActor`, which adds

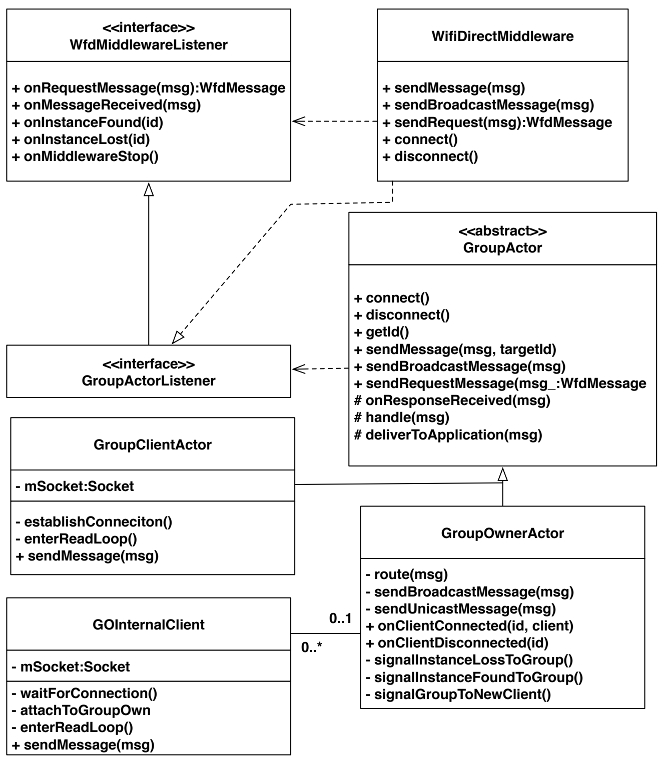


Fig. 4. Design of the WFD_2_SPF middleware.

the methods needed by regular group members to establish a connection with the group owner, and to send it messages. Finally, we have class `GroupOwnerActor`. It also extends `GroupActor`, and adds the methods that are specific to the group owner, such as those needed to route messages, and to react to clients that connect and disconnect to the group. `WifiDirectMiddleware` contains all the logic needed to coordinate discovery, advertising, and the creation of the star overlays, while `GOInternalClient` contains and manages the socket that connects a simple group member to the group owner.

C. Lessons Learned

AllJoyn requires that all interacting devices be connected on the same local area network. This means that the SPF cannot be used in scenarios where a common Wifi network is not available, e.g., scenarios involving user mobility. Moreover, due to an Android limitation according to which it is impossible to activate both the Wifi and the 3G/4G networks at the same time, if the Wifi network being used by AllJoyn does not provide Internet access, the user will be able to use the features of the SPF but will not be able to connect to the Internet.

The Wifi Direct version of the SPF overcomes the above limitations. However, while implementing this version, we also discovered that WiFi Direct has important scalability issues. The maximum number of simultaneous peers that can be managed within a single group is very low (e.g., seven or eight). We attempted to solve this by having lazy peer connections to groups. In practice, the idea was to always keep the amount of peers that are actually connected to the group below the actual upper bound, and perform the connections on demand. This however would have required dynamic and fast peer discovery, which was exactly what we did not have when we implemented the search functionality. Since the star-topology overlay is based on the WiFi Direct groups, our solution has the exact same scalability issues.

These experiments have informed our future work. Indeed, we will continue to investigate the use of the SPF in infrastructure-less scenarios, by studying the adoption of LTE Direct and by introducing new and more complex overlay topologies that could help us bypass the current limitations of WiFi Direct.

V. Evaluation

This section presents the results of the experiments we carried out to evaluate the SPF. First, we evaluated the effectiveness of the framework in facilitating the development of new services/apps. Then, we evaluated its performance and the delays it introduces. Further evaluation is part of our current and future work.

A. Code Quality

The first experiments aimed to evaluate the impact of the SPF on the complexity of the code of a new proximity-based service. As example service we considered a chat app and compared four different implementations, that use or do not use the SPF.

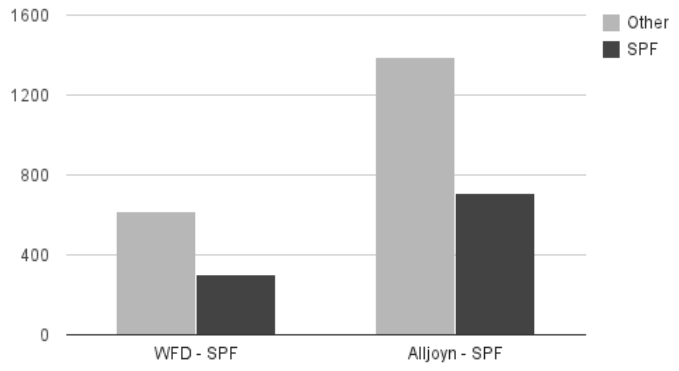
To evaluate the SPF-AllJoyn implementation, we compared the existing chat application bundled as an example in the AllJoyn SDK with a similar application, which we developed, based on SPF-AllJoyn. The two apps offer similar, but not identical, behaviors: the chat built on top of pure AllJoyn offers a group based chat, where users can create groups, join them, and send messages to all the people in the same group. The chat we built on top of SPF-AllJoyn only creates a group on the basis of proximity and sends messages to all current members.

To evaluate the SPF-WiFi Direct implementation, we compared the WiFi Chat app bundled in the Android SDK samples with a new app, which we developed, where the connectivity layer was replaced by a new one based on SPF-WiFi Direct. For a more accurate comparison, the part of the code related to the application's UI was not modified.

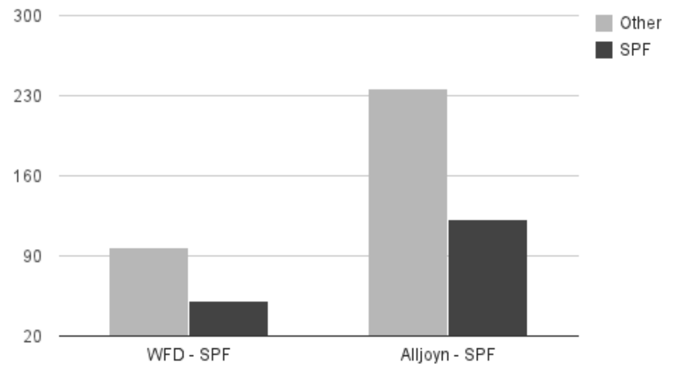
To evaluate the code of the different applications we used SonarQube[7], an open source platform to measure code quality. We focused on two important metrics: the overall number of lines of code, and the cyclomatic complexity, which counts the number of different paths in the source code and provides 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 simpler implementation is easier to debug and maintain.

Figure 5 shows the results of the experiments. Both for the Wifi-Direct and AllJoyn implementations the adoption of the SPF resulted in a reduction of 50% of both the lines of code, and complexity. It also suggested that developers need to

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



(a) Lines Of Code



(b) Cyclomatic Complexity

Fig. 5. Results on code quality.

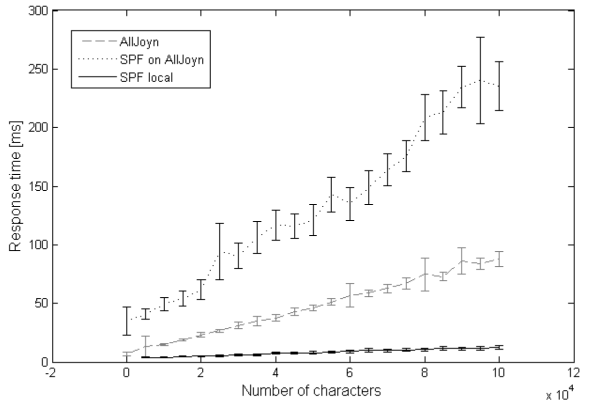
implement and maintain less and simpler code, and they can thus spend more time on other core parts of their applications. Note that the improvement provided by the framework is related to the network layer and clearly shows how Wifi-Direct and AllJoyn, without additional layers, are too general-purpose to be directly used for implementing social-proximity services.

B. Performance

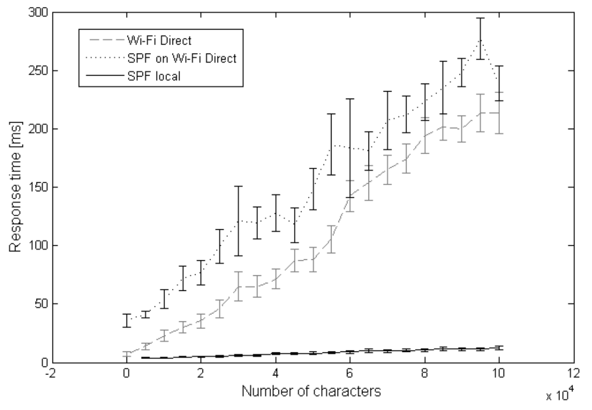
We then evaluated how the introduction of the SPF affected the performance of the apps that rely on it. The experiments aimed to collect the response times of different implementations of the same service/app that differ for the adopted communication layer.

More in detail, we created a ping service that sends requests and waits for acknowledgements from the receiver. The different sizes of the payload were obtained by providing a random 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 6 shows the results of the experiments. If we compare the response time of a local transaction against the one of



(a) SPF-AllJoyn



(b) SPF-Wifi-Direct

Fig. 6. Results on response times.

a remote service invocation, as expected, the impact of inter-process communication is negligible with respect to the duration of a network transaction. If we analyze the results with and without the SPF, we discover that the current implementations of the framework cause extra delays on the response time of up to 50ms for the Wifi-Direct implementation and up to 200ms for the AllJJoyn implementation. 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. The difference between the response times of the two configurations with the SPF is 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. In this case, we found that the framework suffers scalability problems due to the intrinsic limitations imposed by the Android IPC (Inter-Process Communication) framework. Indeed, as described in the API reference documentation, 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. We conducted a set of stress tests by sharing media contents. The usual messages and pictures used on the web and on traditional social networks do not suffer this problem. However the problem occurs when the payload contains a 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.

VI. Related Work

The high availability of smart devices has led to a recent proliferation of different approaches that provide abstraction layers to support proximity-based communication and service provisioning [4], [5], [6], [7], and mobile social computing [8], [9], [10], [11], [12], [13], [14], [15] (described in a survey paper [16]).

AllJoyn [4], as described in Section III, provides a cross-platform middleware for connecting a wide variety of smart devices. Peers in AllJoyn are applications —and not devices— that interact with each other by leveraging a shared distributed bus architecture. The architecture of AllJoyn is a backward compatible extension of D-Bus, a Linux inter-application communication protocol that favors interoperability, re-usability, and security for various language bindings [17]. Although the middleware is flexible and supports different application domains, higher-level services focus on IoT applications and leave the burden of managing social interaction to the developers of specific applications. The introduction of the SPF facilitates the work of developers and favors the creation of new, more sophisticated social proximity services, also on top of AllJoyn.

ShAir by MIT Media Lab is another promising middleware [6]. It differs from AllJoyn in the possibility of creating opportunistic proximity-based networks that can automatically adapt when new devices appear or existing ones disappear [18]. As for AllJoyn, the social interaction is currently not provided. The integration of the SPF with ShAir can then be beneficial for the development of social proximity services in highly-dynamic spaces such as those in public transportation.

Intel Common Connectivity Framework (CCF) is a proprietary middleware developed by Intel Corporation providing a cross-platform transport-neutral proximity-based connectivity [5]. Intel CCF includes many of the features provided by AllJoyn, and it also includes the notion of *user identity* for service advertisement and discovery. Intel CCF proposes a first attempt to introduce user identity as a first-class object. However, the social profile proposed by the middleware is limited to mainly a username and an avatar. Preferences and interests are not supported and thus they cannot be exploited for discovery. Moreover, the identity management is centralized on a single proprietary cloud-based solution. This choice precludes the possibility of using CCF when no Internet connection is available.

Crossroads [19] is a framework proposed by Microsoft Research for developing proximity-based social interactions. The framework shares with the SPF the same problem but it addresses it from a different perspective. Indeed, the framework works at network level and focuses on energy efficiency, topology robustness, and group dissemination load. MobiClique [9] is another middleware that leverages the social profiles exposed by mobile devices via bluetooth to build and maintain an ad-hoc network to provide users with the reliable exchange of contents. Differently from these works, the SPF relies on an abstract communication layer and focuses on providing higher-level primitives to develop diverse social-proximity services.

As the SPF, MobiSOC [14] is a middleware that promotes the social interaction among mobile devices. The focus of the middleware is to provide a shared social state of all the users in proximity. However, the proposed approach requires the presence of regular servers distributed over the space where thin mobile clients can interact with. The SPF, and especially its WiFi-Direct implementation, relies on a lightweight communication layer that does not require centralized solutions.

VII. Conclusions and Future Work

This paper fosters the convergence between new interconnected devices and old-fashioned social interactions. A *social smart space* is a (physical) place where members — humans and appliances— can interact and exploit the services provided by the others in a technology-mediated way. Physical proximity, a key enabler to let the members of a physical space exploit the "services" provided by others, is used here to customize the behavior of the smart space and tailor the actual interactions and provisioning of services. Technologies also help the different members customize their behavior according to different identities and exploit them based on habits and needs.

The paper proposes the use of a *Social Proximity Framework* (SPF) as the underlying infrastructure for the creation of social smart spaces where individuals can interact and cooperate according to physical proximity, and according to the identifies they want to reveal. Two different versions of the SPF, which use diverse communication platforms, were used to demonstrate the key characteristics of the solution and explain how the SPF can exploit the new connectionless communication protocols. Both a first assessment and our industrial partner are positive about the work done.

Our plans for the future comprise the extension of the functionality provided by the SPF, its porting on different mobile operating systems, the development of further and more sophisticated applications on top of it, and also a thorough assessment by our industrial partner for future commercial exploitation.

Acknowledgments

This work has been partially supported by Telecom Italia S.p.A., Strategy & Innovation / Open Innovation Research, Joint Open Lab S-Cube. The authors would like to thank Telecom Italia for the support and help provided throughout the different phases of the work.

References

[1] D. Camps-Mur, A. Garcia-Saavedra, and P. Serrano, "Device-to-device communications with Wi-Fi Direct: overview and experimentation," *Wireless Communications, IEEE*, vol. 20, no. 3, pp. 96–104, June 2013.

[2] "LTE Direct workshop white paper," https://www.qualcomm.com/media/documents/files/lte-direct-whitepaper.pdf, 2012.

[3] T. D. Camacho, M. Foth, and A. Rakotonirainy, "Trainroulette: promoting situated in-train social interaction between passengers." in *UbiComp (Adjunct Publication)*, F. Mattern, S. Santini, J. F. Canny, M. Langheinrich, and J. Rekimoto, Eds. ACM, 2013, pp. 1385–1388.

[4] "A common language for the internet of everything," https://www.alljoyn.org/.

[5] "Intel common connectivity framework," https://software.intel.com/en-us/ccf, 2014.

[6] D. J. Dubois, Y. Bando, K. Watanabe, and H. Holtzman, "Shair: Extensible middleware for mobile peer-to-peer resource sharing," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 2013, pp. 687–690.

[7] E. Toledano, D. Sawada, A. Lippman, H. Holtzman, and F. Casalegno, "Cocam: A collaborative content sharing framework based on opportunistic P2P networking," in *Consumer Communications and Networking Conference (CCNC), 2013 IEEE*, Jan 2013, pp. 158–163.

[8] S. B. Mokhtar, L. McNamara, and L. Capra, "A middleware service for pervasive social networking," in *Proceedings of the International Workshop on Middleware for Pervasive Mobile and Embedded Computing*. ACM, 2009, pp. 2:1–2:6.

[9] A.-K. Pietiläinen, E. Oliver, J. LeBrun, G. Varghese, and C. Diot, "Mobiclique: Middleware for mobile social networking," in *Proceedings of the 2Nd ACM Workshop on Online Social Networks*. ACM, 2009, pp. 49–54.

[10] M. Basuga, R. Belavic, A. Slipcevic, V. Podobnik, A. Petric, and I. Lovrek, "The magnet: Agent-based middleware enabling social networking for mobile users," in *10th International Conference on Telecommunications, 2009. ConTEL 2009.*, June 2009, pp. 89–96.

[11] S. Kern, P. Braun, and W. Rossak, "Mobisoft: An agent-based middleware for social-mobile applications," in *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*. Springer Berlin Heidelberg, 2006, vol. 4277, pp. 984–993.

[12] D. Brooker, T. Carey, and I. Warren, "Middleware for social networking on mobile devices," in *21st Australian Software Engineering Conference (ASWEC), 2010*, April 2010, pp. 202–211.

[13] D. Kalofonos, Z. Antoniou, F. Reynolds, M. Van-Kleek, J. Strauss, and P. Wisner, "MyNet: A platform for secure P2P personal and social networking services," in *Sixth Annual IEEE International Conference on Pervasive Computing and Communications, 2008. PerCom 2008.*, March 2008, pp. 135–146.

[14] A. Gupta, A. Kalra, D. Boston, and C. Borcea, "Mobisoc: a middleware for mobile social computing applications," *Mobile Networks and Applications*, vol. 14, no. 1, pp. 35–52, 2009.

[15] D. Bottazzi, R. Montanari, and A. Toninelli, "Context-aware middleware for anytime, anywhere social networks," *Intelligent Systems, IEEE*, vol. 22, no. 5, pp. 23–32, Sept 2007.

[16] A. Karam and N. Mohamed, "Middleware for mobile social networks: A survey," in *45th Hawaii International Conference on System Science (HICSS), 2012*, Jan 2012, pp. 1482–1490.

[17] "Dbus overview," https://pythonhosted.org/txdbus/dbus_overview.html, 2012.

[18] D. Dubois, Y. Bando, K. Watanabe, and H. Holtzman, "Lightweight self-organizing reconfiguration of opportunistic infrastructure-mode WiFi networks," in *IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems (SASO), 2013*, Sept 2013, pp. 247–256.

[19] C.-J. M. Liang, H. Jin, Y. Yang, L. Zhang, and F. Zhao, "Crossroads: A framework for developing proximity-based social interactions," in *Mobiquitous*, December 2013.