Abstract

This deliverable will describe what a SOCIETIES 3rd Party service is, the features it provides and its structural aspects. A definition of a service from SOCIETIES viewpoint is first provided, the aim being to support the extension of the SOCIETIES platform with services that support interaction and collaboration between members of communities, and exploit the intelligence provided by the pervasive environments of these members. We illustrate the concept through a set of use cases. Then, a review of the current State of the Art relevant to that aim, including platform utilisation through services, pervasive computing concerns and social computing concerns, is presented. Finally, we specify the main functional and information components that are related to the management of 3rd party services. We also propose a set of guidelines regarding the management of the service versioning and dependency.
Disclaimer

This document contains material, which is the copyright of certain SOCIETIES consortium parties, and may not be reproduced or copied without permission.

In case of Public (PU):
All SOCIETIES consortium parties have agreed to full publication of this document.

In case of Restricted to Programme (PP):
All SOCIETIES consortium parties have agreed to make this document available on request to other framework programme participants.

In case of Restricted to Group (RE):
All SOCIETIES consortium parties have agreed to full publication of this document. However this document is written for being used by <organisation / other project / company etc.> as <a contribution to standardisation / material for consideration in product development etc.>.

In case of Consortium confidential (CO):
The information contained in this document is the proprietary confidential information of the SOCIETIES consortium and may not be disclosed except in accordance with the consortium agreement.

The commercial use of any information contained in this document may require a license from the proprietor of that information.

Neither the SOCIETIES consortium as a whole, nor a certain party of the SOCIETIES consortium warrant that the information contained in this document is capable of use, or that use of the information is free from risk, and accept no liability for loss or damage suffered by any person using this information.

Impressum

[Full project title] Self Orchestrating Community Ambient Intelligence Spaces
[Short project title] SOCIETIES
[Number and title of work-package] 3 System Architecture
[Document title] D3.1 Service Model Architecture
[Editor: Name, company] Matteo Bordin, Soluta.Net

Copyright notice

© 2012 Participants in project SOCIETIES
Executive Summary

The SOCIETIES project aims at researching and creating the groundwork for a platform that merges the pervasive with the social, providing capabilities that allow the development of intelligent, adaptable, context-aware, community-oriented services. The framework by itself has limited value to general class of users including users and 3rd party service developers. Building services on top of it leverages the innovations and capabilities of SOCIETIES and expands on its core functionalities and features.

This design document explores the concept of these services, which we define as 3rd party services, which, created by 3rd party developers, extend the core SOCIETIES platform and bring added value to the end-users. A general definition of what is understood as 3rd party service for SOCIETIES is presented, as well as the state-of-the-art relevant for our work scope, i.e. platform utilisation through services, pervasive computing concerns and social computing concerns.

The SOCIETIES platform must provide components and functionalities to support the existence and development of these services. These components, such as the service marketplace and the service registry are presented and analysed, and the interaction between the SOCIETIES platform and a 3rd party service is detailed. The lifecycle of a service, from development to deployment and removal, is described along with its implications on the platform. The service information model (or manifest) is defined, and issues such as versioning, dependency management and development kits are considered.

As a result, a service architecture model for 3rd party services in SOCIETIES is reached, which will guide both the development of these services but also will impact the design of the platform itself.
List of authors

<table>
<thead>
<tr>
<th>Company</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soluta.Net</td>
<td>Matteo Bordin</td>
</tr>
<tr>
<td>SINTEF</td>
<td>Jacqueline Floch</td>
</tr>
<tr>
<td>SETCCE</td>
<td>Mitja Vardjan</td>
</tr>
<tr>
<td>TSSG</td>
<td>Alan Walsh</td>
</tr>
<tr>
<td>PTIN</td>
<td>Sancho Rêgo</td>
</tr>
<tr>
<td>PTIN</td>
<td>Christopher V. Lima</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Other</td>
</tr>
<tr>
<td>4</td>
<td>Service Manifest</td>
</tr>
<tr>
<td>5</td>
<td>Service Marketplace</td>
</tr>
<tr>
<td>6</td>
<td>Service Registry</td>
</tr>
<tr>
<td>7</td>
<td>Service Stack</td>
</tr>
<tr>
<td>7.1</td>
<td>Client Service Development Kit</td>
</tr>
<tr>
<td>7.2</td>
<td>Open Core Service APIs</td>
</tr>
<tr>
<td>7.3</td>
<td>Core Services</td>
</tr>
<tr>
<td>8</td>
<td>Service Life Cycle</td>
</tr>
<tr>
<td>8.1</td>
<td>Publication</td>
</tr>
<tr>
<td>8.2</td>
<td>Deployment</td>
</tr>
<tr>
<td>8.3</td>
<td>Activation</td>
</tr>
<tr>
<td>8.4</td>
<td>Deactivation</td>
</tr>
<tr>
<td>8.5</td>
<td>Removal</td>
</tr>
<tr>
<td>8.6</td>
<td>Deletion</td>
</tr>
<tr>
<td>9</td>
<td>Conclusions</td>
</tr>
<tr>
<td>10</td>
<td>Bibliography</td>
</tr>
</tbody>
</table>
1 Introduction

This Deliverable reports the work done in Task 3.2 Service Model Architecture of the EU FP7 project SOCIETIES.

The aim of this task is to develop the service definition and behaviour, in a technology-agnostic way. It has to clarify the concept of a 3rd party service, how it contributes to the fulfilment of the requirements and use cases created in Work Package 2 and how it enriches the SOCIETIES platform. The service model identifies its features with the dependencies in order to guide WP4, WP5 and WP6 in the development phase.

The architecture presented in this deliverable is part of the general system architecture of the SOCIETIES platform and must be considered in combination with the Deliverables D3.3 (SOCIETIES System Architecture) and D3.2 (Interoperability architecture) so as to provide a complete notion of the project’s platform architecture.

This task is the result of collaboration among six partners (Soluta.Net, Lake, TSSG, Sintef, PTIN and IBM); the work was carried out by means of face to face meetings, weekly audio conferences. UML models where regularly shared using Enterprise Architect.

The starting point of the work carried out in this task, and in the entire WP3 System Architecture, are the scenarios and the requirements extracted by the work-package WP2 (Requirements and Scenarios) and presented in the Deliverables D2.1 (Specification of Initial Scenarios and User Requirements), D2.2 (Scenario description, use cases and technical requirements specification), and D2.3 (Business Requirements, use cases and modelling report).

One of the main goals of this document is to define the concept of a 3rd party service and to investigate the dependencies with the rest of the architecture. An important aspect in this work was to provide a good description of a 3rd party service, in order to highlight how a service can be easily deployed and consumed.

On one hand, we define aspects related to the business model, such as:

- core information, i.e., properties about the service
- interaction information, i.e., how the service can be consumed
- participant information, i.e., roles of businesses and / individuals that become involved in the provision and consumption of a service
- supporting, buying, selling process, and pricing
- trust and reputation of the developer
- legal information, i.e. legal terms and conditions under which the service may be consumed
- Service Level Agreements, i.e., levels of service provided, e.g. availability, response time, etc.

On the other hand, we present aspect related to the service lifecycle and realisation:

- the service life cycle, development, publication, deployment and retirement
- the complex data types that need to be used when extending the platform core data types
- versioning and dependency
Based on the above, the main target of this deliverable was to provide a concrete and complete first description of what is called a “3rd party service” in the context of the SOCIETIES project, in order to highlight what each service offers and how it can be easily deployed and consumed.

A glossary of the main terms used in the project and in this deliverable is available at http://www.ict-societies.eu/glossary/.

Note:

In the rest of the document extensive use is made to a “user”. Unless it is otherwise specified, the term user refers to the class of end-users who ultimately participate with the SOCIETIES platform.

1.1 Services in SOCIETIES

A service is a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description. The consequence of invoking a service is a realisation of one or more real world effects, these effects may include:

- Information is returned in response to a request for that information.
- A change of the shared state of some defined software entities occurs.
- The state of the physical world is affected.
- Some combination of the above takes place.

In SOCIETIES we distinguish between different kinds of services:

**Core Services** are functional capabilities exposed by the platform itself. They are currently drafted in the working document WD31 as part of the functional architecture (Model.SOCIETIES.WP3.Functional.Services) and will be in the future fully described in the Deliverable D3.3 (SOCIETIES System Architecture). They include services for CSS management, CIS management, context management and privacy management.

**3rd Party Services** are an extension mechanism that can be used to add applications which interact with, or extend the services of the SOCIETIES platform. They are not associated with the team that develops and maintains the SOCIETIES platform, but rather provide an opportunity for independent service providers to plug-in value-added services to the platform. As the focus of the SOCIETIES platform is to provide support for the management of communities and the interaction with the pervasive environments of the community members, we expect that 3rd party services will primarily relate to the collaboration within communities and the interaction intelligent environments. The services defined in the Deliverable D2.2 (Scenario description, use cases and technical requirements specification) show a great variety of services that might be added to the platform:

- Register to a business event
- Reserve a taxi for a group of people (include features as number of persons per taxi, cancellation, sharing costs)
- Manage a car pool
- Receive payment (include features such as provider identification and payment method)
• Analyse facial expressions and gestures
• Detect earthquakes or fire events
• Initiate Emergency response based on trigger events
• Get support from volunteers in emergency response

**External Services are** services that are developed independently from SOCIETIES and have no technical or functional dependency with SOCIETIES. On the other hand, External Services can be consumed and used by 3rd party services. Possible examples or external services are:

- geographical maps
- Facebook and other Social Networks (SN)
- Gmail services
- weather forecast
- train and flight timetables
- traffic information.

### 1.2 Support for the 3rd party Services

SOCIETIES is not the first platform that provides extension support through services. Similarly, other social networking platforms also provide support for extension. For instance, Facebook provides an extension mechanism for "Apps" (i.e. applications), a concept similar to services in societies. Another example is that of LinkedIn, but here the extension support is limited to Ads, announcements, and linking and embedding; it does not support extension with new behaviours. Also mobile computing platforms support for extension is provided. For instance, iOS and Android define APIs allowing the development of apps by 3rd party developers and support for extension set requirements on the platform. Typically, the platform should provide the following mechanisms:

- Service lifecycle management supports the deployment, activation and publication of the service on the platform as well as its removal.
- External APIs provide access to the core functionalities of the platform, thus allowing 3rd party services to exploit and build upon the platform capabilities.
- Marketplaces support end-users in order to retrieve information about services, buy and install them.

From a platform point of view, 3rd party services should expose information about themselves such that the platform and also consumers of the service can manage and understand them. This information is typically provided by the service manifest. While the definition of external APIs relates to the specification of the functional platform components and thus is presented in Deliverable D3.3 (SOCIETIES System Architecture), the other issues are presented in this deliverable.

A main challenge is to provide an extension support without jeopardizing the platform, bringing in instability, and allowing the execution of malware and resource overuse. 3rd party services will be run under the control of Access Control (enforcement point) and Purpose Based Access Control that given the declared intentions will monitor their behaviour.

So far, the definition of 3rd party services and the overall mechanisms required from the platform...
to support extension are similar to that of any other service platform. The SOCIETIES platform differentiates from other platforms by its scope, that of merging pervasive computing and social computing to provide support to communities. A community differs from a group of people in that the members of a community share resources to support and maintain common interests and to achieve common goals. The concept of **service sharing** is thus central in SOCIETIES. We consider different kinds of sharing. On one hand, sharing supports the members of a community to share resources of their pervasive environments, either information related to the environment or actions that can be performed in the environment. 3rd party services make use of these resources and the SOCIETIES platform should provide support for sharing these services. On the other hand, the community requires other types of 3rd party services that will allow the collaboration between members in the community. The sharing of these services allows members to perform common activities and achieve common goals. The concept of sharing will be illustrated and further discussed in Section 2.

1.3 Changes with respect to the previously release

This document is a revision of D3.1 previously released in M18. The following changes were made:

- The concept of 3rd party service is revisited with focus on the particular concerns in SOCIETIES. The introduction was revised and a new section that describes use cases is added.
- The section related to the State of the Art was revised to focus on service aspects relevant for SOCIETIES such as platform utilisation through services, pervasive computing concerns and social computing concerns.
- The presentation of the UML model is restructured to facilitate understanding and the semantics of the UML elements is provided, where this was not already done.
2 Use Cases

The term service seems to be one of the most confusing terms in the software industry. This confusion comes from different interpretations made in different conceptual viewpoints such as business viewpoint, end-user viewpoint or computational viewpoint, and also from the different fields for the application of services such as e-commerce or pervasive computing. In SOCIETIES, we also propose to exploit services to support different kinds of extensions of the platform. On one hand, we aim at supporting collaboration within communities. On the other hand, we aim at extending the community’s social space with the pervasive space of each community’s member. In this section, we provide some use cases that illustrate different models of extensions. These use cases introduce informally the platform mechanisms that will be specified later in the document. Being concrete, they aim at establishing a common understanding of 3rd party services and service extension mechanisms in SOCIETIES.

2.1 CSSs and CISs

The discussion of extension to the platform requires the understanding the two main concepts introduced by SOCIETIES:

- A Co-operating Smart Space (CSS) represents a single participant (user or organisation), and includes their information, and services within a distributed collection of CSS Nodes. It provides both a pervasive capability and a social networking capability in an integrated form. A CSS can be associated to zero or more Community Interaction Spaces (CIS), which are a representation of multi-participant community. A CSS can also interact, communicate, or share directly with another CSS, not mediated by a CIS.

- A Community Interaction Space (CIS) represents a pervasive community and has one or more CSSs associated with it. It includes one or more administrating CSSs, a dynamic membership list of member CSSs, a set of shared services/resources, a unique identity, name and description, membership criteria (can be empty for open/public communities), optional community centric information such as preferences, intent models, context, etc.

A CSS has thus the ability to extend the capabilities of the platform with 3rd party services and to share them directly with other CSSs or within a CIS. All services shared in a CIS are associated to a CSS, either a member of the CIS or a administrator of the CIS. The association between a service and a CSS is important as the usage of a service requires the establishment of a contract between the service user and the service provider.
2.2 Cloud-based 3rd party services

In this section, we consider 3rd party services that can be made available to the community members through software installed and operated in the cloud. Typically, these services relate to information sharing in a CIS and coordination in a CIS. They may rely on information captured from the user context.

Use case:

- The organiser of a conference sets up a CIS for conference participants. He configures the CIS with a set of services, among them the “taxi reservation service” allowing people to share a taxi.
- The participants to the conference (their CSSs) join this CIS. They can browse and access the services shared by the CIS.
- To reserve a taxi, a participant uses the shared “taxi reservation service” to specify the preferred pick-up time, pick-up location and destination and send a reservation request. To access the service, the participant simply clicks on the link provided in the CIS.
- The service registers the participant request. It compares it to other requests and, if possible, merges it with other requests. It sets up a group of travellers, makes a reservation to the city taxi service (external service) and sends a notification to all members in the team.

Setting up the platform:

- The conference organiser may either set the SOCIETIES platform himself or use a SOCIETIES platform provider specialized in conference events. The services of the platform are available as cloud software services.
- We assume that various 3rd party service providers develop SOCIETIES-compliant 3rd party services for conference events and publish them through marketplaces. The conference organiser uses the browsing support of the platform to browse among available services in marketplaces and select the relevant ones for the event.
- The SOCIETIES platform provides support for deploying new 3rd party services on the platform, associating them to the organiser CSS and activating them.
- The SOCIETIES platform provides supports for sharing the services in the CISs of which the CSS is owner or member.

Accessing services:

- Access to a shared service requires the CIS participants to install a service client. This is typically done through a web browser or a light-weight desktop or mobile app. The SOCIETIES platform provides support for retrieving and installing service clients.
- Access to a CIS service typically requires credentials. The platform provides support for sharing such credentials.
- Access to the CIS service may also require signing upon an agreement. The platform also provides support for such.
2.3 Pervasive 3\textsuperscript{rd} party services

In this section, we consider 3\textsuperscript{rd} party services that make use of the functionality provided by a smart space. First, we clarify the concept of "smart space". We first present different definitions of a smart space. This is important in order to understand the motivation of the users for tailoring the functionality of a smart space i.e. extending the functionality of a CSS with 3\textsuperscript{rd} party services. The use cases illustrate how services are discovered, how they can be shared with other users, and what is needed for users to access these shared services.

2.3.1 Smart spaces: some definitions

The literature describes a variety of smart spaces, for example:

- A Personal Smart Space (PSS) in addition to traditional smart space management, is self-improving and capable of pro-active behaviour. [Roussaki2010]. This definition highlights the pro-active behaviour of the space. Pro-activeness and autonomy are important properties of smart spaces where a main goal is to let the users concentrate on their tasks rather than on space computing activities that support their tasks. However the shift of focus from computing activities to users’ tasks can also be achieved otherwise, for instance through “natural” interfaces smoothly integrated to the user behaviours and their tasks.

- Many examples in the literature relate smart spaces to limited physical spaces, e.g. a meeting room, a restaurant, an airport. This is inline with the initial experiments in ubiquitous computing as introduced by Weiser [Weiser1991]. Limited physical spaces are a natural way to introduce smart spaces to users: people are familiar to them and thus they are easily understandable.

- When mobility is introduced, smart spaces boarders vanish and become the surrounding of the user. In [Geihs2009], the user is represented by a handheld device that acts as a core to manage the devices in the space surrounding of the user. The space (and the services and devices it provides) change dynamically as the user moves and adapt to the changing user’s needs. The boarders of a space are defined by the technical limits set by the network infrastructure and the discovery protocols.

- Personal smart spaces also consider dynamic spaces controlled by the user [Roussaki2010]. In addition, strong focus is set on the tight relationship between user and space: the space can move with the user, and on communication with other users: the user space facilitates interactions with other user spaces.

- Another kind of user-centric smart spaces are spaces dedicated to specific tasks or goals. For instance, the user might be equipped with a set of medical sensors [Kulkarni2007]. These spaces usually add domain-dependent requirements upon more generic space functionality.

- Smart spaces are usually not solely physical. They also include software services independent of physical objects or locations, and accessible from any place. In an extreme case, we may also consider purely virtual spaces that relate to a goal rather than a location. For instance, a virtual work place gives access to all tools and services useful to perform work tasks, and a virtual meeting place gives access to tools needed to organize meetings.

The variety of definition and application for smart spaces indicates that flexible support is needed to define and organise these smart spaces. The concepts defined in SOCIETIES can be applied in different ways to organize these smart spaces. This will be illustrated by a number of
uses cases in the following paragraphs. We also foresee that the 3\textsuperscript{rd} party services added to a CSS are not solely available through the Cloud (as illustrated by the “taxi reservation service” in the previous section).

### 2.3.2 Smart spaces related to a physical space

In this section, we illustrate a physical space instrumented with embedded devices (i.e. sensors or actuators) with the case of a smart house. We have identified three main scenarios for the configuration of such a space using the SOCIETIES main concepts. They are illustrated in Figure 2:

- An user equipped with a CSS-enabled device entering the space discovers all devices and services in the space, retrieve the 3\textsuperscript{rd} party services needed to control them from a marketplace, and include them in his own CSS.
- The smart house is realised as a CSS (of the smart-home owner with a separate domain identity). A user entering the smart space discovers the smart house CSS and interact with it directly (and the 3\textsuperscript{rd} party services it connects to).
- The smart house is represented by a CSS (of the smart-home owner, with a separate domain identity) and a CIS. The CSS supports the platform utilisation with 3\textsuperscript{rd} party services and the control/sharing of services through the CIS. A user entering the space discovers the smart house CIS and joins it, or a user may be invited to join the CIS. As member of the CIS, the user can interact with the devices share in the CIS.

![Figure 2- Applying SOCIETIES concepts: Smart house representations](image-url)

The first scenario requires the user CSS to discover each device and to download the drivers and applications for interacting with the devices from a marketplace. The discovery is different from the cloud-based case in the previous section: the discovery is no longer realised through a search in a marketplace on the Internet but requires making use of pervasive discovery.
mechanisms, such as wireless discovery protocols or object tagging. Another difference from the cloud-based case is that interaction with the devices (through the SW downloaded from the marketplace) may also be based on short-range wireless protocols. Thus the services provided are only available to users in the proximity of the devices.

In the second scenario, device discovery and interaction is simplified through the house CSS (the house CSS performs similar steps as described in the first scenario). The user CSS now discovers the house CSS instead of each device. This can either be done using pervasive discovery mechanisms (in a similar way as in the previous scenario) or through Cloud-based discovery mechanisms. The latter requires the house CSS to be connected and registered in a Cloud platform. The house CSS shares services with the user CSS directly (i.e. no CIS is involved). That sharing implies that the SW downloaded by the house CSS to interact with the devices also acts as services for the other CSSs, i.e. they provide software interfaces accessible by CSSs (or more precisely client applications extending the CSSs). That scenario presents several advantages compared to the previous one:

- The house CSS hides the interactions with devices, and can provide a uniform interface for interacting with the different devices.
- The house CSS can act as a resource control node in the case there are several users in the house.
- Access to the services shared by the house can be implemented through the Cloud if the house is connected to the Cloud.

Finally, the third scenario adds functionality to the second one. Beyond interaction with the smart space, it allows the users of the smart house (e.g. a family) to interact with each other, and it allows the users to observe the interactions of other users with the smart house devices. In the case the user’s CSS also manage a personal space, services in the personal space can be shared with the community.

The second and third scenarios are preferable for providing access to a physical space. They decrease the complexity of service discovery for the user and enable access through the Cloud. From a service model architecture viewpoint, we see that the platform should support different kinds of discovery mechanisms, one for services available through the Cloud (Cloud-based discovery) and another one for services available in the proximity of the CSS (pervasive discovery). Similarly access to services may occur through the Cloud (Cloud-based interaction) or directly through short-range communication protocols (pervasive interaction). Note that, for a particular device, pervasive discovery may be combined with Cloud-based interaction. For instance, the pervasive discovery may enable the CSS to identify the device and the service allowing to interact with the device available in the Cloud.

### 2.3.3 Smart spaces related to structured physical spaces

In this case, we consider large physical spaces instrumented with a large number of embedded devices, e.g. large buildings owned by an organisation. In that case, structuring the space facilitates management of the space and access to the relevant devices. The user in a particular space needs not generally access to all devices. For instance, the employee of a company typically wishes to interact with the devices of his office when working in the office, and the devices of a meeting room when attending a meeting (or when preparing for a meeting). Figure 3 illustrates two scenarios for structuring the space:

- The building/organisation is represented by a CIS. Each room is represented by a CSS and
is member of the CIS. In that way the CIS maintains the cohesion between the rooms.

The building is represented by a single CSS (and possibly one or more CISs as in Case a). The CSS provides support to create and manage smaller space entities.

**Figure 3- Structuring a smart space**

The first scenario can easily be mapped to the concepts of SOCIETIES. However, there is little opportunity of load balancing between CSSs, eg. Meeting room is full of people (user CSS's), but lunch room is empty.

The second would require the introduction of a CSS structuring concept, that of “space”. A CSS cannot currently achieve this on its own. However, each “space” can easily be represented as a CIS eg “meeting room” CIS, to achieve the desired “structuring”, for administration or control purposes.

From a service architecture model viewpoint, the same issues as discussed in the previous case yield: both pervasive and cloud-based mechanisms should be considered.

### 2.3.4 Other kinds of smart spaces

The different scenarios fit either alone or combines the other kinds of smart spaces that were introduced in section 2.3.1:

- In the case of mobile environments where the space of interest for the user changes dynamically and its boarders are not fixed, it is then difficult to represent the space of interest by a single concrete CSS or CIS entity. We should then consider a mixed approach where some services/devices are discovered by the user CSS, while some might be provided through CSSs/CISs.

- The case of personal space was already discussed in section 2.3.2. We suggest to represent the space by the user’s CSS including a collection of CSS nodes. As the user is mobile and transitions between other spaces, it is relevant to structure the personal smart space to distinguish between “the user private space”, i.e. the devices carried by the user (and thus available most of the time), and “the space around the user”, i.e. the devices that come and go when the user changes surrounding.

- The case of goal-oriented space is quite similar to the previous cases. Again we see a need for structuring spaces, here according to the goal rather than location. For instance, all devices related to the health of the user may be grouped.

- The difference between physical spaces and virtual spaces is two-fold: 1) the mechanisms used for discovering devices/services in physical spaces and services in virtual spaces are different; pervasive protocols cannot be applied in virtual spaces, 2) interaction with virtual services has no impact on physical objects. Otherwise we can consider similar representations of spaces as in the previous cases. For instance, the services of a virtual workplace can be managed as the services provided in a company physical space and the tools of a virtual meeting in a similar way as the services of a physical meeting room. The difference between virtual and physical spaces relies in the nature of the tools/services provided in the spaces rather than in their representations.
2.3.5 Sharing services in a smart space

Sharing services in a smart based may be realised in two different ways:

- The device/service can be accessed directly from any CSS with which the service is shared. The user’s CSS should download a service client and get the credentials in a similar way as in the Cloud-based case, sharing of services in a smart space can be realised.

- The device/service can only be accessed through the intermediary of the CSS that include the service. This means that the user CSS should include some software SW that acts as a service to access the device/service. The user CSS thus becomes a service provider for that device/service. An important concern here is also that the CSS might be a smart phone that is not exposed as a server node in Internet. This requires different mechanisms service interaction to be provided.

2.4 Summary

As illustrated by these use cases, Cloud-based services and pervasive services set different requirements to the SOCIETIES platform. The service model architecture should accommodate both cases. The main differences relate to:

- Service discovery and communication protocols
- Scenarios for the management of spaces (and services in spaces)
- User as a service provider
3 State of the Art

3.1 Software-oriented architecture (SOA)

Several research projects have previously described the State-of-the-Art for service-oriented architecture. This section is based on the State-of-The-Art done by the SIMS and MUSIC projects that some SOCIETIES partners have contributed to [Carrez2007, Devlic2007]. Since it is not the aim of SOCIETIES to make progress within the State-of-the-Art for SOA, but the review made by these projects is sufficient to establish a common understanding of the concept of service within the project and to develop a service model architecture.

SOA is an evolution of component-based architectures. In component-based development, applications are assembled using components written in different programming languages and by different development environments, and running on different platforms. Component-based architectures address the problem of software reuse: a component is a unit of reuse. SOA goes a step further addressing the problem of business service reuse. The Organization for the Advancement of Structured Information Standards (OASIS) defines SOA as the following [MacKenzie2006]:

A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.

This definition depicts two main elements of SOA:

• Services (or distributed capabilities) may be under the control of different ownerships. SOA defines the role of Service Provider. A Service Provider is responsible for the creation and management of services.

• Services are published, thus enabling the discovery of services. SOA defines the role of Service Consumer. A Service Consumer uses service publication information to discover services and may access services without any knowledge about the system that provides the services.

Thus SOA differentiates from component-based architectures:

• While the Service Provider in SOA is responsible for the creation and management of services (and thus of the systems and components providing these services), the owner of a component-based application is responsible for the instantiation of components.

• Service Provider and Service Consumer are bound together through a contract. Beyond service interface properties, a service description in SOA should describe conditions about using the service, including properties like availability, performance, scalability and security. SOA service descriptions thus extend service descriptions defined in component-based architectures that usually restrict themselves to static interface descriptions.

A service-oriented architecture is not tied to a specific technology. It may be implemented using a wide range of technologies, including REST, RPC, DCOM, XMPP, CORBA or Web Services. The key is independent services with defined interfaces that can be called to perform their tasks in a standard way, without the service having previous knowledge of the calling application, and without the application having or needing knowledge of how the service actually performs its tasks [SOA1].
Some important characteristics of services are:

**Services have software contracts.** The idea of design by contract was pioneered by B. Meyer in [Meyer 1997]. Under this approach, two objects agree on a contract which not only includes the class methods signature but also the surrounding context in which the invocation is made. Preconditions and invariants upon method entry must be preserved. In turn post-conditions and invariants are asserted at the method exit.

**Services have behaviour.** Extending the notion of contract, services have a behaviour that should be published as part of the contract. It is not enough to specify the actual invocation and preconditions. The details of how the service should be accessed should be also made available. This could be realized as additional documentation, preferably in machine-readable form to allow dynamic composition of services.

**Services are encapsulated.** The outside view of a service is its contract. The only observable behaviour for a service is expressed in the contract. There is a complete separation between the service contract and its implementation (encapsulation).

**Services share a message bus.** Services usually communicate with each other by sending messages over a message bus. The concept of Enterprise Service Bus (ESB) denotes an abstraction layer on top of an implementation of a messaging system. An ESB tries to remove the coupling between the calling service and the transport medium [SOA2].

**Services are loosely-coupled.** Loose coupling describes an approach where integration interfaces are developed with minimal assumptions between the sending/receiving parties, thus reducing the risk that a change in one application/module will force a change in another application/module.

**Services are stateless.** By stateless services, we mean that service should not be placed with the burden of holding additional context information in a longer transaction. This makes services reusable in any other context. On the other hand, stateless services can be a serious performance bottleneck in a number of situations. For example, sending a security certificate with each request would be a serious burden for both any consumer and provider. It would be much quicker to establish a session between the consumer and provider, replacing the certificate with a shared token shared.

**Services are dynamically discovered.** Services generally register using a service locator (or registry). A client software may use this locator to find the services it needs to perform its tasks. Client software should take into account that services may come and go at any time, especially in ubiquitous computing environments. A number of service discovery protocols can be the foundation of service discovery like SSDP (Simple Service Discovery Protocol) as used in the UPnP Standard, UDDI (Universal Description, Discovery and Integration) for the W3C Web Services, Jini for Java objects, WS-Discovery, a technical specification by BEA Systems, Canon, Intel, Microsoft, and webMethods Inc.

**Services have a life cycle.** Each service has its own life cycle independent from other services. Services may be started, stopped or restarted without stopping a whole system. W3C has published an informative note [SOA3] describing a typical web service life cycle as a set of nested observable states. Both Up and Down super-states can be decomposed in sub-states.

**Services can be composed of other services.** In a similar way as component composition, complex services can be assembled from simpler services and become itself a service. Composition rules specify both the order and conditions under these services may be invoked. Two possible approaches for composition are known as service orchestration and service
choreography. Under service orchestration, it is assumed that a central coordinator (orchestrator) is responsible for invoking and combining all sub-activities comprising the complex service description. The second approach, does not assume a central coordinator. Instead, the overall service activity is achieved as the composition of peer-to-peer interactions among the collaborating services. The most prominent examples of languages of each approach are WS-BPEL 2.0 for orchestration and WS-CDL for choreography. While these support static composition, dynamic composition is still a topic of research.

These characteristics, but not all, are relevant for the service model architecture in SOCIETIES:

- The concept of contract is important as it supports understanding what the service is about. The contract is part of the service manifest.
- The specification of the service behaviour is a complex issue. Instead we have chosen to specify a dependency to a service client (i.e. a behaviour the client should support).
- Dynamic discovery is important for supporting smart spaces. Devices/services are discovered as users enter smart spaces. This was illustrated in Section 2.
- Lifecycle is important as third party services are used as a means to extend the platform. This was illustrated by the case of CIS configuration in Section 2.
- Encapsulation and loose coupling are important. They are the main basis for the project platform architecture. The service manifest is also a means to hide details of implementation.

Since our goal is not to create a service framework that supports the creation of software systems though the assembly of services but rather to exploit the concept of service for realise the extension capabilities of our platform, we have chosen to not address service composition in our work. However, in order to facilitate the correct deployment of services, we support the specification of dependencies of a service on other services.

### 3.2 Service models

Service models are abstract descriptions of the properties, behaviour, and provenance of service, within a well-defined set of use cases covered by the service model. This is the primary distinction between different service models. There are machine readable, and not intended as a replacement of the user guide or manual. They are useful at different conceptual viewpoints. While they support the consumer in understanding the capabilities provided by a service and setting up an agreement with the service provider (business viewpoint), they allow the platform to manage the service (computational viewpoint). Despite the popularity of the service-orientation paradigm, there is today no widely accepted definition of services. Based on various existing definitions and different service description languages, Quartel et al. have proposed a conceptual framework, COSMO [Quartel2007]. COSMO defines terms, provides general concepts for modelling and reasoning about services, and supports operations such as discovery and composition. In their work, Quartel et al. also proposes to exploit COSMO as a common semantic meta-model and shortly explain the mapping of COSMO with a variety of service-related languages including specification, implementation and analysis languages. In this section, we summarize some of the COSMO main concepts. Because of its generality, these concepts have been useful to us when specifying the service manifest presented in section 4. This framework is however abstract and we have also consider engineering approaches as presented in the next sub-section (3.3) when specifying the service manifest and the other parts of the service model architecture.
3.2.1 Service abstraction levels

COSMO defines a service as the establishment of some effect through the interaction between two or more systems, where usually one is playing the role of service provider and the others play the role of service user. The concept of the asymmetrical roles of “user” (also called “consumer”) and “provider” are widely adopted in service engineering. COSMO does not however exclude other models. For instance, the alternative view of peer or symmetrical roles has also been proposed [Kraemer2009, Floch2010]. In that view, any of the involved systems can take initiative in a service, and a service results from collaboration between peers rather than from an invocation from a user to a provider. Based on this definition, COSMO distinguishes three levels at which a service can be modelled:

- Single interaction: A service is modelled as a single interaction between a service user and provider. The model defines the value (or goal) requested by the service user and the value (or capability) offered by the service provider. This model can be used, e.g., to specify or analyse that the goal of some user and the capability of some provider should match.

- Choreography: A service is modelled as multiple related interactions between a service user and provider. Thus, the model defines the external behaviour that is requested by the service user and that is offered by the service provider. This model can be used, e.g., to specify or analyse interoperability between the service user and provider.

- Orchestration: The service offered by some service provider is modelled as a composition of other services. Typically, the model defines the service provider as a coordinator, which interacts with other service providers and combines the values obtained in these interactions to offer some added value to its user. This model can be used, e.g., to specify or analyse a possible implementation of the offered service.

As explained previously, we will not provide support for composition. Our approach considers multiple interactions between a user and a provider. We also provide support for multiple user (or client) roles, i.e. a service may provide different capabilities to different users. We do not provide support for describing detailed user behaviours, but rather specify a dependency to one or more service clients.

3.2.2 Service aspects

Orthogonally to these abstraction levels, COSMO proposes a classification of the properties that need to be modelled. These properties are structured into service aspects:

- Structure: The structure is concerned with modelling the interacting systems that provide or use services, and their interconnection structure. It includes the ports or interfaces at which services are offered.

- Behaviour: The behaviour relates to the activities that are performed by systems, as well as the relations among them. It consists of the interactions between the service’s provider and its users, as well as their causal dependencies or ordering in time.

- Information: The information is concerned with modelling the domain of systems. The value of a service is established through the exchange of information that has to be interpreted in terms of the domain model of the interacting systems.

- Goal: The goal is concerned with modelling the goal or value of a service.

- Quality: The quality is concerned with modelling the non-functional characteristics of
services. These qualities often play an important role in the selection of services. It is generally impossible to specify one service aspect without referring to the other aspects. For example, to specify certain quality characteristics one must refer to the behaviour, and in order to describe the behaviour, it is usually necessary to refer to the information.

Our model focuses on the service structure, goal and quality. Behaviour and Information are represented though dependencies on software components.

### 3.3 Service Registry

While the previous section (3.2) presented a conceptual framework, this section focuses on engineering approaches. Currently there are a vast number of services available on the web which carry out varied tasks that range from simple requests to complex implementations. These services are also implemented in a variety of languages, using a wide range of protocols. The process of discovering and selecting relevant services is greatly facilitated by the adoption of a service registry, designed to allow developers to register their services and for anybody, developer or user, to locate services that are deemed “relevant” according to a variable set of criteria.

The concept of Service Registry is one of the fundamental elements of service-oriented architectures (SOAs) and is a software component that provides the functionalities of a store for information relating to provided services which the user/client can search as required and re-use any existing service as appropriate.

As detailed by González et al [GAGO09], there are enterprise service–based solutions which involve different types of service. It is also noted by them that any technical concerns can be taken care of at the implementation level, where the actual business logic is implemented. Three main types of services can be identified as follows:

- A business service (BS), used by client applications.
- An application service (AS), is consumed by a BS as above.
- A business-service extension (BSE), which can be consumed by a BS to operate on different AS responses.

Figure 4 shows the main relationships between the elements of an enterprise service–based solution, it shows the description of the three different types of service and their relationships.

The service registry organises the service information, persists the information in the service repository and makes it available to the end users. It also supports the interaction among enterprise applications that provide and consume services. The basic functionality of these registries can be enhanced with the addition of the likes of dependencies management and service versioning. Figure 5 shows how the information is organised in the service registry.
Figure 4- Service Types

Figure 5- Organisation of service registry information
One example of a Service Registry being used throughout the business world is IBM WebSphere Service Registry and Repository. WebSphere provides scalable and automated capabilities such as:

- Records details about services you use.
- Discovers existing services direct from their host environment.
- Adds metadata to the services to describe who use them and what for.
- Makes the service Public in order to promote services reuse.
- Track service use using contract management.
- Version control using service lifecycle management

Another example of service registry in action is detailed in [OSR], Oracle Service Registry, according to Oracle it’s service registry provides a ‘DNS’-like reference for SOA, a standards-based interface for infrastructure to discover and bind to relevant service end points and Governance solution that bridges the gap between the design time and runtime. It also lists amongst its benefits hot-pluggability to support heterogeneous services, standards based service discovery and end to end SOA Lifecycle Management.

Embrace service registry [ESR] shows another example of a practical service registry in use. The EMBRACE Service Registry is a collection of life-science web services with built-in service testing. This registry is set up mainly for the use of EU projects EMBRACE, BioSapiens and ENFIN, but is open for use by all interested parties. It primarily sets out to be a BioCatalogue which sets out, registers and and validates web service relating to the biosciences area of interest. It also includes as part of the information relating to services a live testing history along with current status of the services. So it can be seen that the service registry acts as a method of an advertisement of related services and a means to detail all its services status and it is planned to have a seamless integration between this and the BioCatalogue.

Another implementation of a service registry is seen at [SFR], which details the Spring Framework which also incorporates a service registry and it’s method of implementation. This OSGi service registry enables a bundle to publish objects to a shared registry along with the service associated properties.

These solutions have served as a basis for defining the Service Manifest presented in section 4 and the Service Registry presented in section 6. An important concern in SOCIETIES is that we wish to support different kinds of services: cloud-based services and pervasive services. While the presented solutions are suited for cloud-based services, pervasive services require more lightweight approaches. A service provided through an App on an Android phone does not necessarily require a back-end service part.

### 3.4 Service Level Agreement

When a CSS shares a service with other CSSs in a community, both parties involved need to first agree on service usage terms. The party offering the service may provide multiple options for those terms. The party that intends to use the service may choose one of those options as the legally binding agreement for service usage.

Service Level Agreements (SLA) is an important part of service lifecycle in any SOA-based system which aims to be deployed in a real business oriented environment. SLA fits somewhere between discovery of the service and the consumption of the service. SLA represents the
means through which the parties involved in service sharing and consumption agree different aspects of the terms of service in question. SLA includes the definition of QoS metrics that the service will need to respect. Furthermore the user acting as a service consumer needs to know how the service will treat his private and confidential information. The service provider needs to be sure that the user will pay for the service and needs to provide information on how he is going to charge for the service.

The application of SOCIETIES creates particular needs:

- To avoid bothering the user pervasive environments where a large number of service advertisements are expected, the SLA should be negotiated in a semi-automatic way.
- The use of smartphones in pervasive mobile environments requires lightweight SLA languages and negotiation process.
- Since services are shared within communities the SLA language should cover the handling of private and confidential information and the description of the community role in the provision of content.
- In scenarios where professional are involved (e.g. enterprise an disaster management), the individual users act on behalf of companies or organisations. This makes reinforces of course the need for SLA, and the user should comply the company policies.

There are a number of machine-readable SLA languages: WSLA, WSML, WSOL, SLAng, RBSLA, WS-Agreement, BCL, X-contract, among others. Most are XML-based and many are primarily designed for Web Services. The use of XML is clearly intended to ease integration with other Web-services technology, such as WSDL or SOAP that are also dependent on XML [Skene2009]. The following discussion is summarized from [Skene2009]:

- WSLA, WSOL, WSML, RBSLA and WS-Agreement all rely on the provision of extensions of some kind to permit the complete expression of an SLA. WSLA, WSML and WS-Agreement provide abstract data types in their schemas to guide extensions. RBSLA and WSOL rely on the use of externally provided ontologies (although the precise requirements for these remain unclear in both cases).
- WSLA, WSOL, WSML, RBSLA and WS-Agreement provide very little support for expressing latency, reliability, or throughput conditions. The expression of such conditions relies entirely upon language extensions, or syntactic elements exist but are not accompanied by semantic definitions of sufficient precision to support the calculation of violations.
- The semantics for RBSLA are incompletely specified. WSOL, RBSLA and BCL have no definitive language specification document, and instead are described in collections of academic publications. Neither BCL nor X-contracts benefit from a publicly available definition of their syntax. Clearly, it is not currently feasible to adopt these languages as the basis for specifying SLAs with genuine financial implications, as the parties to these SLAs would have no strong basis for arguing for any particular interpretation of the SLAs in the event of a disagreement.

It is not the scope of this document to select the most SLA appropriate approach for the platform. What the service architecture model provides is the ability to describe SLA information in the service manifest and to include SLA negotiation in the service lifecycle.
3.5 Direct communication technologies

Direct communications or peer-to-peer communications (P2P) are technologies that enable an entity, for example an user, to puts in communication two or more devices without the needed of a central server or a particular pre-configured network infrastructure. They are important for the communication with services in a smart space. This section presents various approaches that can be used by the SOCIETIES platform. It is however not the scope of this document to select the most appropriate approach(s) for the platform. This will be done in WP4.

Many applications take advantages from P2P technologies such as:

- Media sharing – sharing videos, photos, news and so on with other entities.
- Chats – enabling people talk to each other.
- Proximal awareness and social services – discovering new people and services nearby.
- Social networks and social gaming – sharing information or whatever and start playing games.

3.5.1 Wi-Fi Direct

Wi-Fi Direct [WFD] represents a technology that allows Wi-Fi capable devices to talk to each other without using a wireless access points in the middle. Wi-Fi direct permits to stay connected to a hotspot and simultaneously communicate with other Wi-Fi direct devices without losing the hotspot connection. This means that the user can create an ad-hoc network using Wi-Fi direct and share the wireless connection on demand with other devices.

Anyway Wi-Fi Direct is designed to use as a sort of quick connection method between, the result is that Wi-Fi Direct is convenient for sharing or exchanging videos, photos, news, information or more general data on demand and in ad-hoc scenarios.

The difference between a common wireless ad-hoc mode and the Wi-Fi Direct consists in the facts that Wi-Fi Direct relies on:

- Higher data rates transfers (on old ad-hoc transfer rate is very poor)
- QoS & Power management protocols (based on Wi-Fi Multimedia - WMM - an interoperability certification by Wi-Fi Alliance [WME])
- Enterprise manageability

Another important fact is that Wi-Fi Direct is IP-based and can work using service discovery (such as UPnP and Bonjour protocols) for retrieving existing devices information or existing services introducing the concept of a zero configuration network [WDF].

At the moment, only few devices like Samsung Galaxy S2 but not iPhone 4S as been certified to work with Wi-Fi Direct. It means that this technology is still young and it needs to grow but it's very promising [WDS].

At the moment, DLNA (Digital Living Network Alliance) is working about integrating the Wi-Fi Direct with DLNA in order to stream and share multimedia contents with DLNA capable devices [DLNA].
3.5.2 NFC

Near Field Communication (NFC) [NFC] is a wireless connection technology allowing two devices or RFID tags to negotiate and establish a connection in order to exchange a small amount of data. Communication occurs through touch because NFC is designed to work in a very short-range space (up to 4cm), so devices should usually “touch” each other to establish the communication.

Google provides a payment method that uses this technology. This Google service is called Google Wallet [GWS] and it works on Android headsets (Android version 2.3 or higher [ANFC]).

NFC can be also used to initiate a communication and then switch the communication on another kind of link such as Wi-Fi or Bluetooth.

3.5.3 Bluetooth 3.0 + HS and Bluetooth 4.0

Bluetooth wireless technologies were designed from the beginning to provide a cheap, low–power, peer-to-peer, ad-hoc wireless link thus eliminating signal wires between devices. Usually when an entity, like a user, creates and connects one or more devices it’s creating a Personal Area Network (PAN) [WFDvsBT].

However in general the Bluetooth technology is not able of high speed: transfer a large amount of data can result quite slow.

That's why Bluetooth, starting from version 3.0, adopted wireless adapters as data link. In this case the Bluetooth technology is used to pair and connect the devices.

3.6 Service Discovery

Service Discovery is the discovery of devices or resources and the services that these entities offer to other devices which have some form of connectivity to the discovered device. The main topologies covered by this definition include:

- devices exist on one network
- devices exist on different interlinked networks, e.g. internet
- devices exist on a grid network
- devices are co-located and communicate wirelessly e.g. WiFi, Near Field Communication (NFC), UPnP and Bluetooth
- devices are connected through mobile ad-hoc networks (MANET)

In general, existing protocols tend to be used in one of the scenarios above with few, if any, covering all.

This section presents various service discovery protocols that can be used by the SOCIETIES platform. It is however not the scope of this document to select the most appropriate protocol(s) for the platform. This will be done in WP4.

3.6.1 UPnP

UPnP [UPnP] is a is a set of networking protocols to seamlessly discover each other’s device presence on the network and establish functional network services for data sharing,
communications, and entertainment. The UPnP architecture supports zero-configuration networking. Some concerns are:

- UPnP has been designed to work inside Local Area Network.
- Broadcast packets used by the protocol to send “events” are valid only inside the LAN.
- UPnP service contents are described as Local resources (Local IP URL to access them).
- UPnP Forum defines a limited list of devices.

### 3.6.2 Peer-to-Peer

Several protocols are available ranging from WiFi and WiFi Direct (yet to be given a public Android or iPhone API), to Bluetooth and NFC (used as an initiation protocol rather than a service discovery protocol). Other protocols that could be used are Zigbee for sensors and low power devices and MANET type protocols for mesh type networks.

### 3.6.3 Internet

There are different service description protocols dealing with Web service discovery:

- Web service registries using UDDI and WSIL. This approach requires service discovery to be initiated by the client and is usually aggregated across multiple registries.
- WS-Discovery for enterprise or office scenarios. For instance, Java-WS-Discovery [JWSD] is the Java implementation of the WS-Discovery specification. The specification defines a discovery protocol based on multicast requests to locate available services.
- Proprietary service registries such as iTunes and Android Market.

### 3.6.4 Local Networks

The Service Location Protocol (SLP) is widely used.

- Bonjour [BONJ] represents the implementation of Zeroconf [ZC] made by Apple. It gathers technologies for services discovering, address assignment and names resolution. Bonjour can locate printers and more generally devices that offer their services on a network: in the most cases a LAN. Printers and these services can be discovered using Multicast Domain Name System (mDNS) records [mDNS]. Bonjour is released as freeware but the mDNSResponder [mDNSR] source code is available under Apache License on Apple developer site. Indeed an open-source Java implementation of Bonjour can be access with JmDNS Software [JmDNS].

### 3.6.5 Other

Pachube: An Internet of Things example of sensor discovery and sharing. Pachube acts as a data brokerage with sensors emitting data streams of information in a variety of formats to be discovered and used by other sites and users.

Facebook’s Project Spartan creates a HTML5 based application store.
iTunes (Apple), Android Market (Google) and Amazon’s Appstore for Android: These and other independent Android stores, other mobile OS platforms such as Ovi (Nokia), serve as application repositories for different mobile platforms. They are each proprietary repositories that require the smartphone’s OS and/or official application to sign-in and allow application browsing, downloading and payments.
4 Service Manifest

Every 3rd party service has a Service Manifest that describes it. It is the single point of composed information for 3rd party services and stores everything that is relevant to fully describe a service. It is needed to support service discovery, deployment and instantiation.

The diagram in Figure 6 represents all the metadata relative to the model viewpoint of a 3rd party service. These details are to be applied and stored by the Service Marketplace.

![Service Manifest Diagram]

Figure 6 - Service Manifest Diagram
We are going to describe each classifier in Figure 6 and Figure 7 to explain its semantics in detail.

**Figure 7 - Interaction Diagram**

**3rd party service remote backend** A non-SOCIETIES Node that is implementing an external service.

**Importance/Relevance for SOCIETIES:** this is an interface and contains the standard infrastructure for the backend of the 3rd party Service. The interface is necessary to define the standards implementation for a backend, the interface guarantee the SOCIETIES platform communication with the service and the data exchange through the service and the platform. This component is necessary to separate the implementation of the service to the SOCIETIES platform, throw this interface the SOCIETIES platform is independent to the service.
implementation and it is not necessary to train the platform to how use the service because all service must have the same interface.

**Presentation, user tier** Definition of a GUI in a mobile platform neutral language.

It supports the use of mobile technology specific code for the client side of the 3rd party service. Nevertheless the SOCIETIES development environment will support a language neutral approach.

These resources can normally be accessed through the native OS APIs but SOCIETIES will use a mobile platform agnostic approach such as HTML5 and/or frameworks such as PhoneGap.

**Importance/Relevance for SOCIETIES:** the definition of presentation layer is necessary to insert into the Service Manifest because SOCIETIES can instantiate the user interface outside the backend of the 3rd party service. The user interface is running at client side and SOCIETIES could be instance for the 3rd party service the UI, the UI communicate with the backend to retrieve manipulate and manage the user data.

This service manifest component is necessary to guarantee the possibility to the SOCIETIES platform to run the service in each platform and OS, SOCIETIES interprets the “User Tier” and could instantiate the user interface in any device with a difference operation system.

**3rd party Data Type** Custom data types that represent specific needs and that does not belong to the Core data types. When adding new CISs to the platform in the production phase of SOCIETIES, there is the need to add new data types that were not conceived at design time.

**Importance/Relevance for SOCIETIES:** all the 3rd party Service could have his custom data type, it is necessary to define this data and deploy into the service manifest so the SOCIETIES Platform can also be used into during the communication with the service implementation.

The custom data type must be defined because it could be used into the societies platform during the UI rendering, during the communication with the service, and to compare a service to another service. The data type describes what the service manipulate, what the service would like to propose to the societies users communities.

**Device** The device required by the service, it specifies also the version and other capabilities.

**Importance/Relevance for SOCIETIES:** the service Manifest must have the definition of the device in which the service can run, so the SOCIETIES platform can run the service in the correct device.

During the scenarios sometime some services as visible only because the users have a particular device (example: the device have a webcam so the user can use the service “video call”; the device have the GPS so user can use the service geo-localisation), so when the developer deploy a service and fill the service manifest field must have to define also in which device the service can be run.

**Legal** Describes the legal aspect of the Service, such as the licence model of the service

**Importance/Relevance for SOCIETIES:** if some services are not free and they need a contract, if a service is open and not need any contract or if some other cannot run in a particular state because it is not legal (example: the service can use a particular encryption that are not legal in some country), the legal component contains all the information about the legal capability of the service.

**MarketPlaceData** Data that are to be used by the marketplace core service at publication time
Importance/Relevance for SOCIETIES: this component it is necessary to describe the data that it is necessary for the MarketPlace. During the publication time the service communicate with the Market Place and describe it self. The MarketPlaceData contains all it is necessary to describe the service into the market place and to inform the user what the service can offer. The classifier also contains how to buy the service, if it is free and contains also a icon of the service and, not mandatory, the screen shot of an instance of a running service.

With this data the 3rd party service describes itself and the market place implementation used the data during the marketing user session.

This component standardizes the communication with the market place and guarantee that all service could have the same UI representation in virtual shop-window.

Importance/Relevance for SOCIETIES: the 3rd party service is made by developer that they are associated to a specific organization, it is important to understand of which organization a service belong to, so the end user can understand if the service could be interesting or not and it is also a way to endorse the service. In some scenario the end user decide to use a particular service because it is associated to the organizer of the concert or the meeting, so the service could be considered thrust.

Plug-in Native platform code extensions. Optional mobile native extensions required by the 3rd party service implementation.

Importance/Relevance for SOCIETIES: the plug-in describe which features it is necessary to guarantee the correct service execution. Some service run because the device have installed a native plugin to interact with the web-came, or some other need the movement sensor to identified if the device is in movement or not. The service manifest contains the dependency with this plugin so the societies platform understand if could be installed the service or not.

Pricing These are all the attributes that pertain to pricing. It can be extended to address if there is a recurring fee, a fixed price, a pay per use. The attributes in this class are used as a precondition for the installation/deployment phase.

The vendor has two options:

• decide if to set the service as free and proceed for payment at service execution
• define a tariff for the deployment service

In the first case he has to deal with the payment process, in the second case the SOCIETIES platform can take care of it.

Importance/Relevance for SOCIETIES: when a service is deployed into a MarketPlace a service could be sells to a user: the pricing definition, describe how to calculate the sales price. SOCIETIES Market place uses this information during the sales negotiation with the end user.

Privacy Definition Policy Describes how personal data will be managed. Requires to ask the user for consent for personal data processing. This could be addressed by EU Funded project Endorse.

Importance/Relevance for SOCIETIES: the 3rd party service manipulates personal data so it is necessary to understand how this data could be manipulates to the service. Some service need the geo-localization of the user, some other need the born date all data that it is owned by the user the used the service. This component describes how to manipulate this data and is it is
possible to redistribute or reuses the data outside the service. For example the scenario “Patient care and Hospital” describe that some data are shared throw 3rd party service, and the sharing it is possible because the two service are thirsted and because the user decide which data could be shared or not using the Privacy policy definition user interface.

**Service Manifest** The concept of the service manifest is used in different technical environments such as:

- Java SE for the deployment of services and application
- Microsoft for side-by-side assemblies or isolated applications
- Android for application development.

The service manifest emphasizes a distinction between a capability, which represents some functionality created by the CIS creator to address a need, and the point of access where that capability is provided. The latter is to be fulfilled by the service registry.

The main information that is described by the service manifest are:

- the deployable code, both client and the middle tier that has to be executed
- declaration of the exposed interface
- declaration of the required SOCIETIES Services

The Service Manifest needs to be serializable in order to be published. The service manifest should be generated by the SOCIETIES service development environment and stored in a registry of the platform. In SOCIETIES, the set of available Service Manifests will be managed by the Service Marketplace component.

**Importance/Relevance for SOCIETIES:** the Service Manifest describe the service, with this component the SOCIETIES platform can retrieve, instantiate, sell, advertise the service, the service manifest it is like the identity card of the service, so the end user and also all societies instance can understand all about the service, from the definition to the implementation.
5  Service Marketplace

The Service Marketplace will act as searchable index for CSSs when looking for services in a well-known Service Store. It will also function as a repository for applications submitted by developers and optionally record and account payments for commercially available applications.

It should allow for the following type of queries:

1. Specific app search
2. Partial app name search
3. By type or category, e.g. game, mapping, recommendation etc.
4. From a recommendation listing, i.e. top 10 per category, newest, most popular

Other types of information that should be available to users browsing the catalogue are:

1. app users' comments and ratings
2. number of downloads
3. app creator descriptions of app functionality, new functionality and issues addressed in latest release, resource permissions required from user if app is downloaded and activated

Given that SOCIETIES is based on the federation principle, service catalogues should have links to and/or provide mirrors to other catalogues.

Figure 8 represents the main concepts related to the service marketplace, which are described in the remaining part of this section.
AppDeveloper This component allows an application developer to add, update or remove an application from a service catalogue. This component should allow a developer to provide information regarding:

1. The application overall functionality
2. New functionalities and issues addressed in the latest release
3. Resource permissions required from user if app is downloaded and activated (Android OS requirement).

This component should also store application developer credentials, public key details and developer profile information to verify and authorise a developer to add and maintain applications in the catalogue.

Importance/Relevance for SOCIETIES: The reputation of both the developer and the requirements of the installable app about accessing potentially sensitive information on the device should be known beforehand to the user, so that he/she will be able to make the proper choice regarding the opportunity to install and run the app. Relevant for the DISCOVER innovation area.

Application user Although the Service Recommendation and Discovery services that act on behalf of a CSS user can search for services within service marketplaces, it is the user who will ultimately control the downloading and installation of a service. The degree to which a user plays in the service installation process will vary depending on the security model deployed by the service marketplace and the target mobile OS.

1. Apple iTunes - Apple iPhones are only allowed to download from their iTunes service repository and Apple carry out security and validation checks on hosted services.
2. Android Market - Hosted services require developer registration when services are first accepted but the end user is responsible for reporting suspected abuses and to validate and check each installed service’s declared permissions and rights to Android resources.

Importance/Relevance for SOCIETIES: The application user’s preferences and intent will be tracked and managed by the SOCIETIES framework components, so it is important for the Service Provisioning Client to accurately profile the application user’s preferences and habits.

BillingPayments This component handles the billing and payments for 3rd party developers who upload their paid applications. This component is listed here for completeness and it is not expected that it will be implemented in SOCIETIES.

DataStore This component provides the persistence layer for the Service Catalogue and will contain:

1. Application files
2. Developer profiles
3. Application descriptions
4. Users’ comments on applications
5. Download history
6. Billing information
7. Update issues and comments

**Importance/Relevance for SOCIETIES**: This component is the core of the Service Marketplace and provides its value to the end users. Technical details about the persistency format are not considered in this WP, but the Data Store will have to be indexed and can be queried in order to be useful. The federation and synchronisation of contents across several Marketplaces is a useful feature that should be implemented to increase the accuracy and completeness of searches for SOCIETIES 3rd Party Services. Relevant for the DISCOVER innovation area.

**SearchEngine** This component handles CSS queries for stored applications and returns the relevant results. At a minimum it should handle the following query types:

1. Specific app search
2. Partial app name search
3. By type or category, e.g. game, mapping, recommendation etc.
4. From a recommendation listing, i.e. top 10 per category, newest, most popular
5. Links to other federated Service Marketplaces

**Importance/Relevance for SOCIETIES**: This component exposes the Service Marketplace content through its interfaces. The bindings of this exposed interface to several clients might be implemented by exploiting several protocols. For example, the communication protocol with a user device (Smartphone) will most probably be different from the communication protocol with a CSS Node, optimising the throughput and reducing the required bandwidth as much as possible. Relevant for the DISCOVER and CONNECT innovation areas.

**Service Marketplace 1..n** A collection of remote service marketplaces that form a federation of service market places. It is expected that each service marketplace will form bi-directional links with other federated marketplaces and allow service recommendation and discovery services to search through the collection as if it was a single entity.

**Importance/Relevance for SOCIETIES**: It is advised to split the single “logical” Service Catalog into a distributed, federated set of Service Market Places. This not only increases the availability of the Service Catalog by avoiding a single point of failure in the architecture, but also provides a better scalability in terms of available 3rd Party Services.

**Service Provisioning Client** This client application will reside on a user's device and handle the service provisioning when a user has selected a particular service(s) to download from a service marketplace.

This client will require the following properties to successfully provision services to users’ devices:

1. Must be part of the SOCIETIES client to ensure that malware substitutes are not used in the download process.
2. Checks that a chosen service marketplace's details are valid and possibly checks its details with a SOCIETIES domain server to prevent man-in-middle-attacks.
3. Conforms to an agreed protocol with the marketplace in requesting and downloading the selected service.
4. Ensures that payable applications have been successfully paid before prior to downloading.

**Importance/Relevance for SOCIETIES**: This component will take care of transparently
performing all the security checks that will be required to improve the security concerns related to the download and installation of external application into the user's device. Ideally, it should not be possible to by-pass the service provisioning client and force the installation of a 3rd Party Service whose credentials and requirements are unknown (just by copying it into the device memory storage, for example).
6 Service Registry

The Service Registry diagram details all the components related to the service registry component and all their associations. It shows the associations to components such as:

- Service Provisioning
- Registry Synchroniser
- Service Browser
- Search Engine
- 3rd Party Service Server

The association to these respective components are also detailed in Figure 9 so as to show the complete picture of all associations to the service registry.

A detailed description of the semantics of each classifier in Figure 9 follows.
3rd Party Service Server

This component represents an external server(s) used for deploying the 3rd party service that will interact with the SOCIETIES platform.

Importance/Relevance for SOCIETIES: Without centralised 3rd party service repositories, the sharing of services becomes problematic with regards to a user wishing to install a 3rd party service and another user(s) wishing to share the user’s service. The provision of services through repositories is widely accepted and has recently become ubiquitous through various app stores.

Centralised distribution also allows efficient upgrading of previously installed services and clients, notifications and possible service recalls in the case of malicious software.

Registry Synchroniser

It is envisioned that there will be several Service Registries, although the Service Browser will only ever query one. Therefore this Registry Synchronizer component will synchronize all Service Registries that it is aware of.

It does this by the use of the events, when the registry is modified by the addition or deletion of a service this will through a registry event, with the required details which will be picked up by other nodes listening for these events and update the appropriate registry with the relevant details.

The potential changes that the registry synchronizer will detect and propagate across registry instances are:

- Service added
- Service deleted
- Service status changed

Importance/Relevance for SOCIETIES: Each node in a user’s CSS will have the capability of installing and hosting services. The purpose of the Registry Synchroniser is to allow local service registries to be synchronised with other CSS nodes. One mode of operation could involve a master service registry with local registries acting as fall-back registries in the case of a node losing connectivity.

Search Engine

This component handles queries for stored services and returns the relevant results. It should handle the following query types:

1. Specific app search
2. Partial app name search
3. By type or category
4. Most popular Apps

As an input to this component it will take the CSSID of the CSS to be searched, the ID of the service being searched for, or the “Type” of service required. This can then query the service descriptions stored in the registry and return the relevant results.

In terms of output from this component, it will return a list of services (service IDs) that match the relevant search requests.
Importance/Relevance for SOCIETIES: Modern App stores allow various modes for a user to explore the apps available in a particular repository. Feedback from users can also be incorporated to give user ratings of services to guide other potential users and developers.

Service Browser
The Service Browser component will provide an interface to the Service Registry. It will allow the querying for a list of available services and an Explorer type GUI to display.

The service browser GUI will allow the user to view available services in the system and filter them based on input provided by the user. It will allow filtering such as:

- Search for a particular service based on the service ID
- Search for services of a particular Type
- Display a list of all services
- Display the most popular services

Importance/Relevance for SOCIETIES: The Service Browser is the public face of the service repository or marketplace and utilises the Search Engine to expose services to different forms of browsing/discovery.

Service Provisioning
Service provisioning refers to the setup and lifecycle control of a 3rd Party service or resource in correlation to a CSS. Further details on the contractual requirements on 3rd Party services can be found in WD3.2 and are not covered here. Service provisioning covers the following capabilities:

- Installation of new 3rd party services.
- Configuration/reconfiguration of 3rd party services.
- Enforcement of sharing policies for 3rd party services, when access is required.
- Removal of 3rd party services

Importance/Relevance for SOCIETIES: This component handles the transfer of a selected app/service to a CSS node, its installation and configuration and removal of services.

Service Registry
The Service Registry component will store the list of Services available to the CSS, on which node they are hosted for request routing and possible access criteria such as whether a service is available to be shared by another CSS in a CIS community.

Importance/Relevance for SOCIETIES: The Service Registry component acts as a CSS store for installed services and allows other services within a CSS to determine if other services are available for interaction. It also allows requests to services to be routed to the hosting CSS node.
7 Service Stack

This section will present an overview of the planned environment provided by SOCIETIES to develop 3rd party services. This environment, the Service Stack, comprises of the Service Development Kit, the Open Service APIs and several core SOCIETIES components. The diagram presents the envisioned service stack.

---

**Figure 10 - Service Stack**

---
7.1 Client Service Development Kit

The Service Development Kit (SDK) supports the development of 3rd party services. The environment makes use of the core services and thus depends on a set of APIs provided by the core platform. In particular, it will give 3rd party developers easy access to the Open Core Service APIs which they may use to access SOCIETIES platform functionality.

7.2 Open Core Service APIs

The Open Core Service APIs are a subset of the Core Services APIs that are made available to 3rd party service developers and providers. Through these APIs developers will have limited access to SOCIETIES platform functionality. By limited access, we mean that not all platform functionality should be exposed to 3rd party developers (for example, for security reasons). By defining a specific subset of APIs that developers may utilize, the integrity of the SOCIETIES platform is reinforced.

The APIs might be specified using a language neutral IDL (Interface Definition Language), e.g. UML classes or MOF. In such a case, these APIs descriptions could be used to generate language-specific code.

7.3 Core Services

There are several Core Services and components that support 3rd party services, either directly by exposing a set of functionality to be used by developers, via the Open APIs, or indirectly by providing background support and capabilities to the platform:

- **Privacy & Trust:** Allows a service to negotiate with the platform on its requirements and levels of access through the use of policies. Access Control acts a gateway to ensure that the negotiated access levels are enforced. It determines how a 3rd party service interacts with the SOCIETIES core services and it controls access to CSS and CIS resources (e.g. user's profile and context). The Access Control component is normally used in conjunction with other SOCIETIES API calls, not directly by the 3rd party service. As Trust Management is concerned, it needs to be decided how 3rd party services are made available to the SOCIETIES platform. For instance, services can be directly available from the service provider through a service marketplace (such as Android Market) that may not provide Trust functionality. Another case is availability from other CSSs (directly or mediated through the CIS). This choice will dictate how Trust is handled by the platform.

- **Context Management:** Allows a service to optionally act a context source and/or use CSS/CIS context information. Context is a key feature for SOCIETIES and the intelligent, adaptable and pervasive services for which the platform is intended. By providing context information to the platform, a service may potentiate richer functionality by the platform or other services to the user; by consuming context from the platform, the service may be able to better adapt to the user’s needs.

- **CSS/CIS Communication:** Provides routing and transport to messages to and from other services which may reside on other CSS devices and/or CSSs. This component is essential to support the pervasive and social nature of the platform and its services. It will allow services existing on one node of a CSS to interact with other nodes or CSS, exchanging data and providing extra functionality. Without it, 3rd party services would only be able to execute on a specific device in an isolated manner.
• **Service Personalisation:** Allows the service to configure itself to accommodate the user's preferences. This may entail setting various parameters and behaviour of the service to those inferred from a user's profile, context or preferences from similar services. The capacity of a service to adapt itself to the user is a crucial component to achieve the goal of having an ecosystem of pervasive, intelligent and pro-active services. This component of the SOCIETIES platform offers functionality to the services that allow them to achieve this goal.

• **Service Lifecycle:** Controls the lifecycle of the 3rd party service from initial installation to its possible deletion. A lifecycle for 3rd party services will be defined; this lifecycle may differ from the core service lifecycle. This component doesn't supply functionality directly to the 3rd party service, but is nevertheless essential for its existence. It will allow the service to exist within the sphere of the SOCIETIES platform, monitor its status and allow platform components to discover it.

• **Service Registration:** This service handles the registration of the service with the host CSS. 3rd party services cannot interact with the SOCIETIES platform without previous recognition by the platform itself. The Service Registration component handles the registration of 3rd party services and allows other components in the platform to know its attributes and characteristics. The 3rd party service's attributes, their metadata, should be described using a standard method (e.g. WSDL, SML, WSMO). Such descriptions will be supplied by 3rd party developers.

• **Event Manager:** Allows a 3rd party service to register events that may be of interest to other services and register as subscriber to core and other 3rd party services' events. This component permits a flexible, event-driven approach to service design and communication. Different services may exchange information in this manner, or adapt their functionality in an intelligent, pervasive manner that potentiates richer functionality. For example, a restaurant recommendation system might register to context events about specific locations, and be automatically informed when the user is near such a location.
8 Service Life Cycle

A 3rd party service that is developed and deployed into the SOCIETIES platform may be in one of several stages, each with its own implications on the platform. The lifecycle of a service represents these stages and the various phases it goes through, as well as the main activities in each one. It helps define the tasks the platform must support to allow the existence of 3rd party services, as well as give a picture of the overall process and behaviour.

A new 3rd party service can be submitted to the platform, starting the PUBLISHING phase, which ends when the service is accepted and made available in the Service Marketplace.

At this point, the service is PUBLISHED, but does not exist in executable form in the platform. A CSS must deploy it; in this process, the DEPLOYMENT phase, the CSS must download the service from the service marketplace and provision it. This will be the task of the Service Provisioning component of SOCIETIES, which will coordinate the entire process and interact with the necessary platform components; for example the Trust and Privacy components to ensure the service to be installed fulfills the criteria of the CSS user. At the end, the service will be in the DEPLOYED state: technically capable of being executed. It should be possible, however, for a service to be “ready-to-execute” but the users not actually be allowed to do so. This marks the difference between the DEPLOYED and the ACTIVE stage of the lifecycle, the transition between then being the ACTIVATION phase.

Finally, for each phase mentioned, there is a reverse: DEACTIVATION, REMOVAL and DELETION. At the end of the last one, the service ceases to exist in the platform.

We are now going to describe in detail each service lifecycle transition by means of Activity diagrams.

8.1 Publication

This phase consists of the publication of the service by the developer, thus giving the platform visibility to it.
**Status: CREATED**

The service has been created, but is not published to the Service Marketplace and thus is not available in the platform.

**Package Service**

This includes packaging the service binaries, as well as the descriptors in the proper form for the platform. This can be done automatically by the SDK, and a complete deployable unit should be the result, as well as the service manifest.

**Publish to 3rd Party Server**

[Diagram: Act Publication]

- **Status: CREATED**
- **Package Service**
- **Submit Service**
- **Validate Service**
- **Validation Tests**
- **Publish to 3rd-Party Server**
- **Status: PUBLISHED**

**Figure 11 - Publication**
A 3rd party service might also have some logic running not on the SOCIETIES platform itself; rather, this would operate on an external 3rd party server that is contacted by the components of the 3rd party service that exist within the platform.

**Submit Service**

Submits the service to the platform. More concretely, the point of entry is at the Service Marketplace.

**Validate Service**

Every service submitted to the platform needs to pass a number of validation tests to determine whether it can be accepted (configuration values defined and default values provided, correct metadata). This task should be done by the Service Marketplace before it accepts a submitted service.

**Status: PUBLISHED**

The service has been created and exists in the platform, at the Service Marketplace level.

### 8.2 Deployment

This phase of the lifecycle entails the provisioning, configuration and actual deployment of the service in the CSS. At the end of this phase, the service should be ready to run.
Get Service Information
This activity consists in consulting the Service Marketplace and retrieving the necessary information and service metadata.

SLA Negotiation
Before a 3rd party service is obtained from the Service Marketplace, an SLA must be agreed on. Each Policy must be checked and agreed on. The Service Provisioning should interact with the Trust & Privacy components of the platform for this negotiation process.

**Check Dependencies**

The CSS (more concretely, the Service Provisioning component) should determine whether it has all the necessary resources and dependencies necessary to run this service. These are specified in the Service Manifest and may be, for example, device requirements or binary libraries.

**Load Service Artifacts**

Downloading the actual deployable units, i.e. code, to the CSS from the Service Marketplace.

**Install service**

The ServiceProvisioning component will install the necessary artefacts in the CSS nodes, in the SRE. Some parts may be running in a client node (e.g. mobile), others in a middleware tier. This activity includes all deployable components, from logic to GUIs.

**Configure Service**

The service is configured according to the characteristics of the CSS and the preferences of the user.

**Register Service**

The service is registered in the Service Registry.

**Deployment: Failed**

The necessary dependencies or policy agreements to deploy the service have not been met. The service can't be deployed and the user should be informed of the error.

**Status: DEPLOYED**

The service artefacts have been downloaded from the Marketplace, fully installed and provisioned in the CSS. However, the service is not available to be executed by the user, until it is activated.

### 8.3 Activation
During this stage of the lifecycle, the service transitions from being deployed to being active and thus able to be executed by users.

**Start Service**

It should be possible for a service to be deployed, but not be running or be able to be executed by the user yet. This activity changes this status.

**Status: ACTIVE**

The service can be executed at will by users.

### 8.4 Deactivation

During this stage of the lifecycle, the service transitions from being active, and thus will able to be executed at will by a user, to deployed. During the deactivation stage, no new requests for service execution may be accepted, while current executions must be terminated.
**Status: ACTIVE**

The service can be executed at will by users.

![Diagram](image-url)

*Figure 14 - Deactivation*
Service being executed?
The service shouldn't simply be stopped from the ACTIVE state, as there might be end-users executing it. As such, a check must first be made to determine if any user is using the service before stopping it.

Finish executions.
Before the service is stopped, current executions should be allowed to terminate cleanly.

Check dependencies
It's possible that other 3rd party services in this CSS depend on this service in order to execute. If this is the case, then a decision must be made.

Service is Shared?
A check must be done if the service is currently being shared. If it is and we mean to stop it, then we must first stop this.

Stop services
Stop all other active services that depend on this service.

Stop Service
Once the service is no longer being executed by any user, it can be stopped, switching back to the Deployed state.

Status: DEPLOYED
The service artefacts have been downloaded from the Marketplace, fully installed and provisioned in the CSS. However, the service is not available to be executed by the user, until it is activated.
8.5 Removal

This phase is the direct opposite of the Deployment phase. It happens when a service is removed from the CSS.

Status: DEPLOYED

The service artifacts have been downloaded from the Marketplace, fully installed and provisioned in the CSS. However, the service is not available to be executed by the user, until it is activated.

Unregister Service

The service is unregistered, so it can no longer be discovered using the Service Registry.

Remove Artifacts

Removes all service artefacts from the CSS that were downloaded from the Service Marketplace and installed by the Service Provisioning.

Status: PUBLISHED

The service has been created and exists in the platform, at the ServiceMarketplace level.

8.6 Deletion

This phase is the opposite of the Publication phase. The service is completely removed from the platform.
Status: PUBLISHED
The service has been created and exists in the platform, at the Service Marketplace level.

Delete service from Service Marketplace
The service is removed from the Service Marketplace, thus no longer being present in the platform.

Status: DELETED
The service no longer exists in the platform.
9 Conclusions

This document has outlined the Service architecture for the interaction of 3rd party services with the SOCIETIES platform. The creation and execution of these services has been discussed, particularly in relation to the possible use of SDKs.

Various aspects of 3rd party services such as their meta model provided by their service manifest, service interactions; the use of marketplace service registries to collect, track their downloading, provide billing and developer monetisation and allow service discovery; and the issues of versioning and dependency management were discussed.

The more platform specific issues such as service lifecycle, service SDK, interaction with SOCIETIES core APIs and how downloaded services will registered with the platform have also been considered.

The outcome of this work will be used in each development work package WP4, WP5, WP6 in order to guide developers in the implementation of the service model described in this document. This work will also be used during integration (WP7) to validate whether the requirements of 3rd party services are fulfilled.

Given the heterogeneous platforms (JVM host OSs and Android), the main challenge will be to support as many types of possible 3rd party service implementations. Apart from the obvious ones such as Android applications and OSGi-based services, it will also be a design challenge to support other possible 3rd party service interactions such as non-Java based services which may be running on remote servers and browser-based services.
10 Bibliography

- [JmDNS] JmDNS official site, http://jmDNS.sourceforge.net/

• [mDNS] mDNS, http://www.multicastdn.org/


• [WESE] WebSphere Service Registry and Repository


