UbiCollab:
A Service Architecture for Supporting Ubiquitous Collaboration

Christian H. Mosveen
mosveen@stud.ntnu.no

Andreas Brustad
andreabr@stud.ntnu.no

Spring 2006

Abstract

Ubiquitous computing integrates computation into the environment, and enables users to move around and interact with computers more naturally than they currently do. This helps to address some of the traditional challenges of computer supported collaborative work (CSCW), as users are not bound to a desk and a personal computer, and are not forced to stay in a static environment where ad-hoc collaboration is impossible. UbiCollab is a platform for the support of ubiquitous collaboration, and it provides such functionality as context-awareness and automatic device discovery. The vision of UbiCollab is to be both flexible and extendible, so that it can provide ubiquitous collaboration support for many different existing and future domains and settings. A previous study has compiled a set of requirements that needs to be fulfilled in order for a platform to reach this vision. This work re-designs the architecture and the platform components of UbiCollab so that they conform to these requirements. OSGi is chosen as the underlying architecture, supporting the requirements of flexibility and extendibility, and a suitable OSGi framework for the platform is chosen. The platform components and their application programming interfaces (APIs) are designed, and a selected number of these are implemented with full or partial functionality. A testbed of applications and external services is used throughout development to test the flexibility and functionality of the platform and the completeness of the APIs.

Preface

This report documents the work done as a Master's thesis in computer science at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) in Trondheim. The thesis is a contribution to the UbiCollab platform, and is performed in the spring of 2006. The project assignment was given by IDI and Telenor Research and Development, and is a continuation of previous work on the same topic.

The report contains work on the architectural analysis and re-design of UbiCollab, forming the basis for the new UbiCollab platform. Following is the task description:

> UbiCollab (short for Ubiquitous Collaboration) is a platform for supporting collaboration using ubiquitous and mobile technologies.
>
> UbiCollab has been developed through iterative design and testing in various projects. Based on the evaluation of the existing platform done during an autumn project, this task will:
>
> 1) propose a new architecture based on newly emerged basic technologies with special focus on OSGi (Open Service Gateway initiative), and
>
> 2) design and develop selected parts of the new architecture as a proof of concept.

We wish to thank our project supervisors, Professor Dr. Monica Divitini and Babak A. Farshchian for insightful input and ideas, along with valuable feedback on the research and development, and the writing of the report.

Trondheim, June 16, 2006.

Christian H. Mosveen Andreas Brustad

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Humans have traditionally collaborated in many ways to perform tasks and achieve goals easier. In the last few decades, this trend has also carried over to computers, and much research has been carried out in the field of computer supported cooperative work (CSCW) [8, 9]. Traditionally, the applied side of this field, known as groupware (mainly a collection of tools suited for one particular type of collaboration), has been focusing on supporting tasks for office work [3, 15]. Usually, the groupware applications have combined such work-related features as video, audio and textual communication, document sharing, and concurrent editing. The main problem with these applications has been that they are very tool-centric - not transparent to the user so he or she can focus on the real task - and that they are not useful outside the domain of office work. Focus has recently shifted away from the traditional office environment, and collaborative systems have now been deployed in new settings such as construction sites [16] and hospitals [1].

Lately, mobile computing has become more and more popular, and is an important step towards more flexible and ubiquitous computing. This mobility has brought computing away from large, stationary desktop computers typically found in offices or similar facilities, and by doing so has extended the support to new categories of users in radically different situations [26]. The world has also changed, so that work that was traditionally done only in the office or at the desk now can or has to be performed while taking the train or at the café. Additionally, it has become more common to use computation for other aspects of life than work, for instance in devices for communication and entertainment, and in almost any other type of electronic appliance. This should also mean that computer supported collaboration can be moved outside of both the office setting [16], e.g. taking part in meetings while on the train, and from the work domain altogether, for instance entering the domain of entertainment or social computing.

Computer supported collaboration is useful in traditional work settings with

fixed groups and clear, common goals, but it can also be influential in other settings where the groups are not so fixed, and the users do not share a common and pre-determined goal. Examples of such settings are for instance some cases of open source development, support systems for nursing homes or households, and social computing involving communication and experience sharing.

To support such a variety of different uses and usage domains, one single application would not possibly cover every needed aspect, and a suite of different applications would be hard to maintain and would mean much duplicated work as most applications would share some common functionality. Instead, having a platform that provide general functionality common to all domains, upon which specific applications can later be built, is much more flexible.

Platforms that are intended to support such a variety of collaborative settings and applications have to be explicitly focused on being general enough to provide the common functionality of all domains and thereby not exclude any of them, while still being able to provide enough functionality to be of practical use to every single one of the given domains. While platforms for collaboration exist, there is none that satisfies this broad goal. UbiCollab was developed to provide such a platform [6], and this work will provide an extended and redesigned version of the platform, bringing UbiCollab nearer its vision.

This chapter introduces the project and its motivation, its goals, and the contributions made by the work. It also discusses the research method used.

1.1 Motivation, goals and contributions

1.1.1 General motivation

The general motivation for this work is to assist in making support for collaboration more ubiquitous and available in a wider array of settings than the traditional office domain. In this work, creating a platform for ubiquitous collaboration is the chosen approach for meeting these challenges. By letting the platform deal with issues such as automatic discovery of available services, management of persistent data storage, and retrieval of context information such as current position coordinates, application developers can focus on implementing the actual functionality of the applications. A prototype of a platform for ubiquitous collaboration, called UbiCollab, has previously been developed by Schwarz [25] and extended by Jensen [14] and Rasmussen and Braathen [22].

In the autumn of 2005, work was done to analyze what requirements were common to a set of different collaborative scenarios, and how well existing platforms for collaboration, including UbiCollab, fulfilled these. That work will throughout this report be referred to as the autumn report [19].

The evaluation conducted in the autumn report allowed to point out some limitations of the old version of UbiCollab with regards to these requirements. This is particularly evident with the overall requirement of being flexible enough to support different collaborative domains and settings, and with the non-functional requirement of being easily extendible whenever new service interfaces or modified functionality is needed. Additionally, the old version of UbiCollab is lacking with regards to some of the functional requirements, such as providing possibilities for persistent data storage, discovering resources on different types of technology, and retrieval and management of different types of context information.

In this work, the requirements found in the autumn report, along with the experiences with the old version of UbiCollab, are used as a starting point to design a new and improved architecture for UbiCollab.

1.1.2 Project goals

The vision of UbiCollab is to provide a flexible and extendible platform for supporting ubiquitous collaboration. Because of the flexibility, it should be possible to use and benefit from the platform in a multitude of different settings and domains. The extendibility should ensure that the platform easily adapts to new settings and scenarios (i.e., by providing clear and purposeful service APIs), that extra functionality is easily added to existing settings (i.e., having services that are easy to extend with new functionality), and that the platform supports the introduction of new and heterogeneous technologies (e.g., new resource discovery or positioning mechanisms).

To accomplish this, it is of essence that the platform supports the arrangement of several different resources, supports mobility, is applicable to several different domains, supports collaboration in various forms, such as both synchronous and asynchronous, co-located and distributed, provides management and awareness of available resources, and supports people management such as privacy and presence [25]. Some of these aspects are covered in other existing collaboration platforms, such as the ad-hoc resource management of Gaia [13], the mobility support of the ABC Framework [4], and the support for various collaboration forms in Collaborator [2], but there are no platforms handling all of them.

The prototype of UbiCollab, made by Schwarz [25], is a step towards fulfilling this vision, but it is not complete. The most critical drawback of the old version of UbiCollab is that it is very focused on office work, and is only tested with applications for remote and co-located meeting support. Another important drawback is that while it was originally designed as several independent components, these components became dependent on each other after development, and the platform became very hard to extend with new functionality.

The goal of this work is to re-design the platform architecture of UbiCollab, so that it conforms to the platform requirements extracted in the autumn report [19]. Of highest importance is designing the platform so that it is flexible enough to be usable and beneficial in different settings and domains, not just office work, and that it is easily extendible whenever new service needs emerges, or modifications to existing services has to be made.

Part of this involves investigating the adoption of OSGi, a specification for frameworks supporting plug-in based components and their interdependencies, as the underlying architecture of UbiCollab. A suitable OSGi framework has to be chosen, and components forming the UbiCollab platform have to be designed. It is of importance that these components are independent of each other, and that they can be deployed in different subsets, or as a full set, depending on the needed functionality in a certain setting or on the capabilities on a specific device.

1.1.3 Contributions

This report will present the following set of contributions made by the work:

- Primary contributions
  - Design of platform architecture
  - Evaluation of designed platform
    - With regards to platform requirements
    - By testbed and demonstration experiences
  - Specification of platform service API
  - Working versions of selected platform components
- Secondary contributions
  - Testbed technology experiences
  - Testbed software instructions and fixes

Primary contributions

A discussion on how the overall requirement of platform flexibility and the non-functional requirement of extendibility from the autumn report [19] are fulfilled by designing UbiCollab to run on top of OSGi, and utilizing OSGi's built-in plug-in mechanisms and dependency handling as the bottom tier in the UbiCollab architecture, is performed. Additionally, the functional requirements from the autumn report [19] are distilled into functional blocks and designed as separate service components, forming the core services of the platform.

The designed UbiCollab platform is evaluated with regards to the platform requirements extracted in the autumn report, and compared to the evaluation of

the old version of UbiCollab. The designed UbiCollab platform is also evaluated with regards to the experiences discovered when testing the platform with several small test applications and services, and from the two platform demonstrations.

The full platform application programming interface (API) is specified by separating the platform functionality into service components and specifying each of these platform services' API.

A set of selected platform services are fully developed and implemented, and these working components form the current prototype of the re-designed UbiCollab. Additionally, these components are used to assess the success of using OSGi as the underlying architecture of UbiCollab, and as the test platform for the different test applications and services making sure that the services' exposed functionality is adequate. The fully implemented platform services include the Collaboration service, the Discovery service, the Positioning service, the Location service and the Data storage service.

Secondary contributions

Practical experiences with developing for smaller devices, such as PDAs, are discussed. This includes documentation of problems that might arise when developing for these kind of devices and suggestions on how to fix these problems.

Instructions on how to deploy and run the platform and the accompanying test applications are given. This also includes detailed software fixes and modifications for certain parts of the testbed.

1.2 Research method

The research method for this work is driven by the design and development of service components that are conforming to the requirements found in the autumn report [19], and continually using the testbed to evaluate the service these components provide. This evaluation may then uncover new functionality the service should provide, or needed modifications on how it provides it, and the process starts over again until the service is in accordance with the requirements, and the functionality tests yield positive results.

The research method can be illustrated as in Figure 1.1.

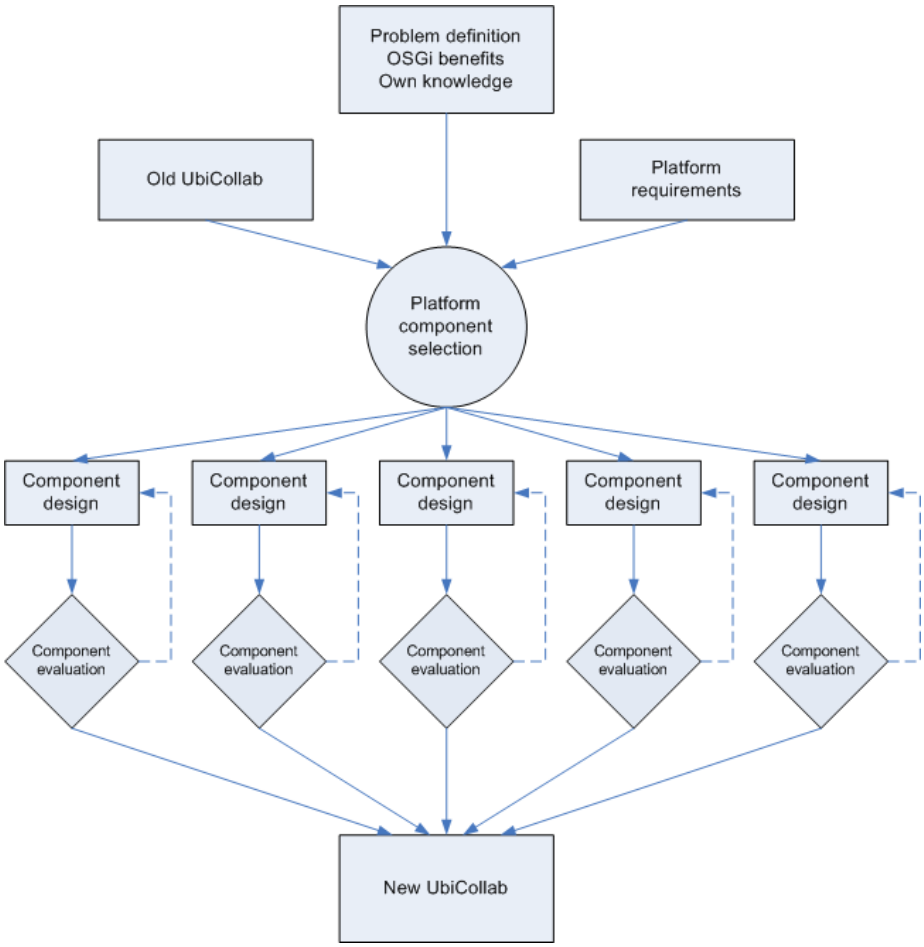

Figure 1.1: Research method

1.3 Report outline

The rest of this report is organized in the following chapters:

2. Problem elaboration This chapter briefly discusses the challenges of ubiquitous computing and collaboration technology, and how UbiCollab fits in the intersection between these. It also discusses the old version of UbiCollab, and the platform requirements extracted from the autumn report [19]. Finally, it presents a set of research questions that this work will address.

3. Underlying architecture This chapter defines the need for platform flexibility and dynamicity, and introduces OSGi as an underlying architecture for the platform to contribute in providing those aspects. It also discusses the benefits of OSGi compared to current alternatives, and presents a list of available OSGi frameworks. Finally, it presents the framework chosen for UbiCollab and how an architecture with OSGi as the lower tier improves UbiCollab.

4. The UbiCollab platform This chapter discusses the new architecture of UbiCollab, and presents the design and overall functionality of the service components chosen to form the platform's core services.

5. UbiCollab implementation This chapter discusses general implementation issues of UbiCollab, and then presents the implementation details of all service components that have been developed. These details include the full service API, and certain key points about the inner functionality. For the service components that are not fully developed, suggestions as to how this should be done are given.

6. Platform demonstration This chapter introduces the testbed used for testing and evaluating the platform during and after development. It also presents each application and service made with the intent of testing different combinations of platform services.

7. Platform evaluation This chapter presents an evaluation of the new UbiCollab platform with regards to the requirements found in the autumn report [19]. It also compares the evaluation of the new platform against the evaluation of the old one. Finally, it analyzes the experiences received when testing the platform with the created test applications and services.

8. Conclusions This chapter concludes the report, and presents an evaluation of whether, and to what degree, the work's goals were reached. It also discusses how the goals were reached, and analyzes any possible limitations in the approach, execution or result. Finally, this chapter presents some thoughts around further work that should be done on UbiCollab.

Appendices Appendix A presents a set of fixes that had to be made to different pieces of software in order to be able to run UbiCollab on a PC and a PDA. Appendix B presents an installation guide for UbiCollab; both for a standard PC and for a PDA (specifically an HP iPAQ).

Chapter 2

Problem elaboration

This chapter briefly discusses the challenges of ubiquitous computing and collaboration technology, and how UbiCollab fits in the intersection between these. It also discusses the old version of UbiCollab, and the platform requirements extracted from the autumn report [19]. Finally, it presents a set of research questions that this work will address.

2.1 Ubiquitous collaboration

2.1.1 Collaboration technology and ubiquitous computing

A unique feature of UbiCollab is the combination of the two domains; collaboration technology and ubiquitous computing.

Collaboration technology

Computer supported collaborative work (CSCW) and the applied side of the field, known as groupware, aims at using technology to enhance the collaborative work of groups [8]. In contrast to single user applications, which are generally used to solve a particular problem, groupware is designed to use the computer to facilitate human interaction. Focus areas for collaboration technology does therefore include communication, collaboration and coordination (see [19, 8] for a more detailed description of the domain).

Ubiquitous computing

Ubiquitous computing aims at integrating technology and computational devices into daily life to an extent where the devices "disappear" and users can utilize them without having to think about how, or even notice that they are there at all. It is claimed that ubiquitous computing is the opposite of virtual reality. Whereas virtual reality invites humans into an artificial computer

generated world, ubiquitous computing tries to bring computers into the "real world" [26]. This means moving away from traditional computational platforms such as laptops, desktops and even PDAs and mobile phones, and onto devices tailored for the particular task (see [19, 26, 17] for a more detailed description of the field).

There are several reasons why it is beneficial to combine these two domains:

Disappearing computing Traditional groupware has introduced a number of constraints on how collaboration is performed. Users will, for example, have to know how to use a computer, connect to the network and start a program in order to see whether a colleague is available for communication. This is far from a real world scenario, where the user would simply knock on the colleague's office door. By combining collaboration technology with ubiquitous computing and its vision of the "disappearing computer", users can focus on the actual collaboration without having to worry about the underlying technology and how to use it [26, 24].

Mobility As collaboration can take place anywhere, computer support for it should also be location independent. By introducing ubiquitous computing to CSCW, such computer support can be possible not just in different locations, but also while on the move [19].

Constantly changing environment In order to better match the real world, one of the focus areas of ubiquitous computing is to move from a static computing environment to a dynamic one. Collaboration is no exception, and takes place in a highly dynamic environment with users logging in and out, changing activity status, etc. Groupware can therefore be greatly enhanced by introducing concepts from ubiquitous computing.

2.1.2 Challenges of combining the two domains

As seen, there are many advantages of combining collaboration technology and ubiquitous computing. On the other hand, combining the two domains does also mean that the platform is faced with challenges from both domains.

Erlich [9] points out an important distinction between single user applications and groupware. Whereas single user applications tend to be designed to support a particular task, groupware aims at supporting the work-process of a group. Such work processes tend to be tacit and amorphous, and are thus more difficult to support. Erlich stresses the importance of studying local work practices and adapting the applications to them rather than the other way around [9]. For a platform such as UbiCollab this has serious consequences. Instead of, for instance, defining a standard way of supporting online meetings, the platform will have to support meetings the way people in the local organization are used to

arrange them. This means that the platform will have to be extremely flexible, and will have to work in a multitude of different settings and scenarios.

Deployment of groupware applications also involve serious challenges [5]. Groupware applications often require sophisticated infrastructure, and will normally have to be installed in several locations at about the same time. As mentioned in the previous section, users should be unaffected by this and should be able to continue their daily work without having to worry about where the system is deployed, what version of the system is currently running and how to run it etc.

The high degree of mobility typically found in a ubiquitous environment presents challenges as well. The need for mobility makes users dependent on wireless communication. Designers will therefore have to address issues like seamless handover between networks, loss of network signal, varying bandwidth, etc [10]. In addition, security and privacy issues become more critical when dealing with wireless communication [23].

Finally, a dynamic and constantly changing environment will force designers to deal with issues like seamless joining and leaving of uses and services, how to make sure the system can keep running if a component fails and how to customize content and communication techniques to the capabilities and constraints of the user's device.

2.2 The UbiCollab platform

The environment in which UbiCollab is supposed to operate will be populated with several different kinds of users working on numerous distinct devices in a multitude of different settings and scenarios. Creating a single application capable of supporting such a variety of domains and users will be extremely hard, if not impossible. Previous work on UbiCollab [19, 25] has therefore concluded that the responsibilities should be shared between two different layers; the platform layer and the application layer. UbiCollab makes out the platform layer, and should focus on providing applications with commonly needed services and resources. In this way, application designers can build their applications on top of the UbiCollab platform and focus on the domain and device specific functionality. Unfortunately, creating UbiCollab as a platform does also introduce some new challenges, and requires some new design and evaluation techniques compared to traditional application design and evaluation.

2.2.1 Challenges in design and evaluation of UbiCollab

As pointed out by Edwards et al. [7], there are good techniques for designing and evaluating applications, whereas the techniques for designing and evaluating platforms intended to support these applications are much less well formed. The problem is that it is hard to design and evaluate the features of a platform without knowing about its applications and users. Trying to anticipate the needs of all possible applications will lead to a platform that is too complex and hard to use. With too few functions, on the other hand, the platform will offer little value to the application designer. In both cases, the effort required to install and understand how to use the platform could potentially exceed the added value of using the platform, which will lead to the platform not being used at all. For UbiCollab to become a success it is therefore crucial to carefully select and evaluate what services and functionality to provide to the applications, to make sure the feature set of the platform is balanced.

The evaluation process is also more complex for platforms than for user-visible applications. The overall goal for any software system is to improve the end-user experience. However, measuring this for an infrastructure system such as UbiCollab is complicated because end-users do not use the software directly. In order to evaluate the framework, applications that use features of the framework have to be developed. As stated by [7], this leads to a number of questions that have to be answered:

- How to choose which applications to build in order to evaluate the framework?
- What does the evaluation of the test-application say about the underlying framework?
- Can we use the same evaluation-techniques to do this "indirect" evaluation as we use for evaluating end-user applications?

2.3 Platform requirements

During the autumn of 2005, an evaluation of the first version of UbiCollab was conducted [19]. The purpose of that project was to investigate how UbiCollab could be improved to support collaboration in several different environments, increase end-user experience and reduce the time and effort required to develop and maintain collaborative ubiquitous applications. The report lists several requirements that UbiCollab have to fulfill in order to accomplish its vision. These requirements are summarized in Table 2.1. For the purpose of this thesis, the non-functional requirements are especially important. This is due to the fact that these requirements to a larger extent influence and define the underlying architecture of the platform. Support for these requirements will therefore have to be taken into consideration from day one, whereas support for the functional

requirements more easily can be added at later stages in the development. This is, however, dependant on having an architecture that is easy to extend with new functionality. Consequently, the extendibility requirement is especially important. Platform flexibility is also highlighted as an important requirement even though it is not explicitly mentioned. Despite the focus on the non-functional requirements, functional requirements cannot be completely forgotten in the initial phases of the development. Without supporting any of them it is very hard to test whether the other requirements are met by the chosen architecture. How can, for instance, the scalability of the platform be tested if the platform does not provide any services? In fact, without support for any of the functional requirements, there might not be any implementation at all, as the non-functional requirements generally relate to how things are done and does therefore not have any meaning on their own. The approach adopted in this thesis has therefore been to focus on the non-functional requirements as well as some of the functional ones. For the sake of completeness, all the requirements, including those that have not been addressed in this thesis, have been summarized below.

Functional	**Non-functional**
Presence management	Scalability
Resource collection	Platform extendibility
Positioning	End-user programming
Location management	Device OS independency
Privacy	Comm. channel independency
Security	
Persistent data storage	
User profile management	
Asynchronous comm. support	

Table 2.1: Platform requirements

2.3.1 Presence management

The platform should be able to store and manage presence information (whether a user is capable of communicating) and availability information (whether a user is willing to communicate) about the users of the system. This information should be specific to the different communication types, so that a user for example could be available for one type of communication while at the same time unavailable for another type.

2.3.2 Resource collection

The platform should be capable of detecting resources that are connected to the network. It is advantageous if the search could span several different middleware,

e.g., UPnP[1], JXTA[2], etc. Upon detection, these resources should be made available in a searchable resource repository so that other users can query the repository for resources that mach certain criteria.

2.3.3 Positioning

The platform ought to be able to determine the position of connected resources and users. There are many available technologies for positioning for instance GPS, GSM, WLAN and the new Galileo system. The platform should support several such positioning mechanisms, so that it can return positioning data from the most accurate one at all times.

2.3.4 Location management

As coordinates are not particularly user-friendly, it should be possible to return the name or description of a location corresponding to a set of coordinates, for example returning "Manhattan" instead of the specific latitude and longitude. The platform will therefore have to be able to store the coordinates compromising the boundary of the location zone and its name, and provide means of testing whether a set of coordinates is contained by any of the stored zones.

2.3.5 Privacy

The users of the system should not feel like they are being monitored when using the platform. In most cases, users should therefore be able to use the system anonymously without having to send personal details to the platform, nor other users.

2.3.6 Security

As UbiCollab is a multi-user system, security is a major concern. The platform will therefore have to enforce authorization for all data and resources in the system. However, this should not come at the expense of user experience, and users should therefore not need to sign in more often than once per session.

2.3.7 Persistent data storage

Many applications will require persistent data storage. As some of the devices in the system might have limited or no storage capacity, this service should be provided by the platform. Having a common data storage space could also benefit certain collaborative applications such as shared workspaces. To make the system as flexible as possible, the persistent data storage service should be platform independent and easy to extend with other features such as version control.

[1] http://www.upnp.org/
[2] http://www.jxta.org/

2.3.8 User profile management

If allowed by the user's privacy settings, the platform should store common details about its users and make this data available to other platform components and users. This will allow applications such as address books to be created, and will enable users to search for other users and retrieve their contact details.

2.3.9 Asynchronous communication support

Communication is essential for any collaborative system. Users should therefore be able to send messages to each other even when the recipient is not online. This is especially important if users are collaborating across different time zones. The platform will therefore have to store messages on behalf of the communicating users until the messages can be delivered. This does also improve robustness of the system, as sending does not have to be cancelled if the recipient loses network connection during transmission.

2.3.10 Scalability

UbiCollab will be found in a highly dynamic environment, with users and resources constantly joining and leaving. It will therefore be hard to determine, at design time, the number of users that will utilize the platform at all times. Consequently the platform will have to scale well, so that performance is kept at an acceptable level even when the number of active users is high. To do so, the traffic through the platform server should be kept at a minimum, unnecessary network transmissions should be eliminated and the server load should be distributed among several servers. In the cases where the user's device has decent processing power, some typical server-tasks could instead be handled by the client's device.

2.3.11 Platform extendibility

Over time, new requirements will appear, user needs will change and new standards will emerge. This will require new platform services, modifications to old ones and cause some services to be rendered redundant. Making such changes to the platform should be as easy as possible. If not, people will not bother to make the necessary modifications to the platform, which will eventually result in an outdated platform that no one will use. The fact that UbiCollab is developed incrementally does also necessitate an easily extendible platform, so that the code clarity, etc., are not lost along the way.

2.3.12 End-user programming

Even though end-users do not interact directly with the platform, the platform design will affect the applications built on top of it, and thereby influence the end-user experience. Many tasks can not be fully automated, and will require input from the user. Configuring the environment to user needs, combining

resources to get the desired effect etc. are examples of such tasks. With a large number of services and resources in the system, and an environment where resources join and leave constantly, this could become a technically advanced and complex job. It is therefore essential that the platform makes these tasks as easy and straight forward as possible.

2.3.13 Device operating system independency

Ubiquitous computing aims at customizing devices for their intended task. This will result in a large number of distinct devices with different capabilities, constraints and operating systems. It is important that all of these devices can collaborate within the system. Consequently, the platform will have to be operating system independent.

2.3.14 Communication channel independency

The platform should allow users on different network types to communicate with the platform and each other. These networks might have different characteristics, such as bandwidth and addressing schemes, and the platform will therefore have to either customize the transmitted content to the characteristics of the actual network, or provide information so that the clients can adapt it themselves.

The study stresses that these requirements will change over time, as new technologies, user needs and standards emerge. It is therefore essential that it is easy to extend the platform with new services, remove outdated ones and modify existing services and components. As the platform will have to work in such a variety of scenarios with different kinds of users and devices, the platform will also need to be extraordinarily flexible. Finally, there are issues and challenges relating to the extreme dynamicity of the environment that are not directly addressed by the above mentioned requirements. These include finding "backup services" if the service in use suddenly fails, spreading services among different peers to reduce the risk of single point failures, and adapting the platform features to the resources and services currently found in the environment.

2.4 Research questions

The overall goal of this work is to take advantage of OSGi to improve the architecture of UbiCollab so that it conforms to the platform requirements found in the autumn report [19], with a special focus on the requirements of extendibility and flexibility. In order to achieve this goal, it is necessary to perform several distinct tasks. First, a new platform architecture based on OSGi has to be designed. Second, the application programming interface (API) of the new

platform has to be given. Third, a working prototype built with the new design has to be implemented. And finally, the new design and prototype have to be evaluated.

Therefore, the research questions this work will address are:

- How can the platform architecture of UbiCollab be improved in order to make the platform more flexible and easier to extend?
- How can OSGi be used to achieve this goal?
- What services and functionality will UbiCollab have to provide to application developers through its application programming interface (API) in order to fulfill the requirements in the autumn report [19]?
- How can a prototype of the new platform be implemented in order to most effectively demonstrate the new platform architecture and its features?
- How should the new platform be evaluated in order to clearly establish whether it has succeeded in reaching its goal of being a flexible and extendible platform for ubiquitous collaboration?

Chapter 3

Underlying architecture

This chapter defines the need for platform flexibility and dynamicity, and introduces OSGi as an underlying architecture for the platform to contribute in providing those aspects. It also discusses the benefits of OSGi compared to current alternatives, and presents a list of available OSGi frameworks. Finally, it presents the framework chosen for UbiCollab and how an architecture with OSGi as the lower tier improves UbiCollab.

3.1 Architecture introduction

3.1.1 Platform flexibility and dynamicity

In addition to the platform requirements from the autumn report [19] presented in Chapter 2, here follows a re-stated description of the overall platform requirement of flexibility and dynamicity.

Since the platform will be deployed in many different scenarios and on many different devices, it should be able to handle a flexible and dynamic environment where a specific feature would not be needed in a certain scenario, or where a particular capability is not present on the platform's device. This could for example be a scenario where persistent data storage would not be needed, or where a device would not be equipped with a GPS satellite receiver for positioning itself. In such settings, the platform should both be installable with a subset of features, and installed features depending on non-present capabilities should continue working, although with limited functionality.

To achieve this, the platform should be component-based, meaning that every functional part of the platform should be created as a component. Then different deployments of the platform could include a different set of components depending on the functionality needed. The important aspect here is to make the components independent of each other, but able to add their functionality

instantly when added to a deployment.

Additionally, instead of having functionality depending on that other components or capabilities exist on the local device, such functionality should search for and retrieve the needed component or capability, for instance through a remote discovery service. Both local and remote components and capabilities should be added to this discovery service, and the most suitable one should be returned when searched for. This would normally be a local one, if such a component or capability exists, otherwise a remote one would be returned. An example of this could be when an application would like to know the name of the location of a certain coordinate. After querying the discovery service for location services, the discovery service would return a local one if such a service existed, otherwise it would return any of the remote location services. The application would treat these the same way, without paying attention to whether it was a local or remote one. The only difference would be a possibility of different sets of locations or different names for them depending on what location service was returned.

3.1.2 Platform essentials

In addition to the overall requirement of platform flexibility and dynamicity, the most important aspects when choosing what to build UbiCollab on were the non-functional platform requirements extracted from the previous work on UbiCollab [19]. Of these, two relate directly to the underlying architecture of the platform; namely "Platform extendibility" and "Device operating system independency." For the first requirement, some sort of plug-in system would be advantageous; so as to relieve the extension developer from touching core platform code and only focus on using the relevant platform interfaces. The second requirement would be met by having the platform run on its own, with all of its components being device operating system independent, or by having the platform run inside a container which itself is device operating system independent. In accordance with the requirement of platform flexibility and dynamicity, as mentioned earlier, the platform should also be able to increase and decrease its number of present components according to what kind of device it is deployed on, and what capabilities that device has available. This would make the platform more ubiquitous and dynamic, adapting to its local environment.

3.2 OSGi introduction

The OSGi Alliance was founded in 1999 to create open specifications for networked delivery of managed services to local networks and devices. The specification, named the OSGi Service Platform Core Specification [20], delivers "an open, common architecture for service providers, developers, software vendors, gateway operators and equipment vendors to develop, deploy and manage services in a coordinated fashion." Frameworks created according to this specifica-

tion manage the installation and updating of OSGi components, called bundles, in a dynamic and scalable fashion. This enables OSGi-compliant devices to install, update and remove OSGi bundles whenever suitable.

Bundles are created just like normal Java applications, with the exception of replacing the usual starting point of a main-method with an activator-class implementing the OSGi interface BundleActivator. Upon completed development, bundles are packaged in a JAR-file, and accompanied by a manifest-file specifying bundle details such as name, version and what packages the bundle imports and exports. This importation and exportation is the key to OSGi's plug-in system and how it handles dependencies. If a bundle is a service provider, it specifies the interface to that service as an export in its manifest. Another bundle can then specify the package-name of that service interface as an import in its manifest, and consume the service as if it were a part of the same Java program. To handle such dependencies in a dynamic environment where the consuming bundle might be present when the bundle providing the service is not, OSGi offers a mechanism of tracking services so that the developer of the consumer bundle can specify suitable actions depending on whether the service is present in the OSGi framework or not.

An example of a normal OSGi bundle maintenance schedule would involve identifying the bundle in question and stopping it, whereby every other bundle being dependent on the stopped one would receive events triggering them to halt their functionality accordingly. This would constitute the plugging out of a bundle, and the needed re-development could commence. After being done with whatever maintenance was needed, the bundle should be rebuilt and plugged into the OSGi framework again. In OSGi terms, this plugging in would only require reloading and starting the bundle from within the framework.

When stopping a bundle, it performs an internal clean-up routine by stopping running threads, etc. It also gets de-registered from the framework's bundle context, meaning that other bundles will no longer be able to find it, and those who had service trackers registered for the particular bundle will be notified of its exit. When starting a bundle, it performs internal start-up routines such as instantiating certain functional blocks, before registering itself with the framework's bundle context. Along with this registry, the bundle normally also specifies a set of properties, such as the bundle language or whether the bundle interface should be exposed as a web service.

3.2.1 OSGi's layered architecture

The functionality of the OSGi specification is divided into five layers [20]:

- Security Layer
- Module Layer
- Life Cycle Layer
- Service Layer
- Actual Services

The Security Layer is an optional layer based on the Java 2 security architecture. It provides infrastructure for deploying and managing applications that need to run in controlled environments. The Security Layer does not define an API for application control itself, but leaves that to the Life Cycle Layer.

The Module Layer defines a modularization model for Java. It has strict rules for sharing and hiding Java packages between bundles, in the shape of import and export statements in the bundles' manifest files. While the Life Cycle Layer provides an API for management of the bundles in the Module Layer, and the Service Layer provides a communication model for the bundles, the Module Layer can be used without these two layers.

The Life Cycle Layer provides a life cycle API to bundles. This API defines the runtime model of the bundles; how they are started and stopped, as well as how they are installed, updated and uninstalled. It also provides an event API for bundles to get information from and control the operations of the service platform. The Life Cycle Layer is dependant on the Module Layer.

The Service Layer provides a dynamic and solid programming model for Java bundle developers, by de-coupling a bundle's service interface from its implementation. This "publish, find and bind" model lets bundle developers defer the selection of a specific implementation to run-time, by allowing them to bind to services only using the interface specification. This implementation selection happens through the framework's service registry, where bundles can register new services, receive notifications on the state of services, or look up existing services. The Service Layer is highly integrated with the Life Cycle Layer.

The Actual Services is the top layer, and consists of all services running on top of the other layers. An OSGi service is defined semantically by its service interface, and implemented as a service object owned by and run within a bundle.

3.2.2 Specification version differences

Due to the desire to separate the specification into layers, and other generally large changes, the specification was completely rewritten between Revision 3 and Revision 4. In addition to the introduction of the layers, some of the most important changes were:

Improved modularization support – The framework now supports loading multiple different versions of the same package.

Added security features – Bundle permissions based on digital signature, finer grained framework administration permissions, and a new BundlePermission class to handle the permission to access other bundles' classes and resources.

Manifest localization – The framework now supports localized entries in a bundle's manifest file, making it possible to change entries such as name or vendor depending on who is running the bundle.

Additional event types – Among the event types added was event types for signaling the resolving and unresolving of bundles.

3.3 OSGi on limited devices

Since OSGi is based on Java, it should be possible to run an OSGi framework on practically any kind of device. In relation to this, the biggest problem is to find a suitable Java version, being mature enough to run on the specific device as well as being new enough to support the needed functionality of the OSGi framework. Today, even commonly available PDAs tend to only run older versions of Java, specifically version 1.3. In this situation, an OSGi framework that is also compatible with Java 1.3 has to be chosen, and it is thus possible that newer frameworks with added functionality have to be discarded. On the other hand, it is certainly possible to develop OSGi bundles conforming to version 1.3 of Java, and in any case, as newer devices get manufactured it is safe to assume that their Java compatibility will also increase.

It is also the case that since the OSGi frameworks are Java programs, they need a relatively large amount of memory available to run. This will exclude devices with very little available memory, typically less than 64 MB, unless OSGi frameworks for exactly such devices are developed.

If a specific device has limited or no storage capacity, it is possible to set up the chosen OSGi framework to retrieve every bundle except the core system bundle from another location on start-up. This loads all bundles into volatile or non-volatile memory, and cleans the memory up when the framework is shut

down, but it requires that the framework has network access to a specified bundle repository.

Because of OSGi's "publish, find and bind" model, it is possible to develop consumer bundles that are very flexible, only associating themselves with the interface of the service bundle. Through the functionality of the Service Layer, services are bound at run-time, resulting in a system that can have several different implementations sharing the same interface. It is therefore possible to provide the same kind of service from any particular device, where only the actual implementation is tailored to the device, and let the consumer bundle decide on this choice of implementations on start-up.

3.4 OSGi frameworks

Following is a short analysis of existing OSGi frameworks, highlighting which OSGi specification they are compliant with, whether they are free, open source or commercial, and their current degree of development activity.

3.4.1 Knopflerfish

http://www.knopflerfish.org/

Knopflerfish is an open source OSGi framework sponsored by Gatespace Telematics. All development is currently focused on Knopflerfish 2.0, with the latest release being Beta 4. Knopflerfish 2.0 Beta 4 is feature complete in all mandatory parts of the OSGi R4 specification and service compendium, and there are only minor items missing from the optional parts. The current finalized version of Knopflerfish is 1.3.4, which is compliant with the OSGi R3 specification.

3.4.2 Oscar

http://oscar.objectweb.org/

Oscar is an open source OSGi framework originally created by Richard S. Hall. The latest release is Oscar 2.0 Alpha 7, which is an OSGi non-compliant experimental release prototyping the extensions to the OSGi R3 specification. The latest compliant release is Oscar 1.0.5, which is compliant to a large portion of the OSGi R3 specification. Oscar has also spawned the Oscar Bundle Repository; a repository for OSGi bundles created for Oscar, but that should be OSGi framework independent. Development for Oscar is discontinued as it has been turned into an Apache incubator project called Felix.

3.4.3 mBedded Server

http://www.prosyst.com/osgi.html

mBedded Server is a commercial OSGi framework developed by ProSyst. The current release is mBedded Server 6.0, which has a full implementation of the OSGi R4 specification.

3.4.4 Equinox

http://www.eclipse.org/equinox/

Equinox is an open source OSGi framework managed by the Eclipse Project Management Committee. The latest release is Equinox 3.2 Release Candidate 3, which fully implements the OSGi R4 specification. Equinox can be used as a standalone OSGi framework, but its main purpose is acting as Eclipse's runtime environment, handling the life-cycle of all Eclipse components and plug-ins.

3.4.5 SMF

http://www-306.ibm.com/software/wireless/smf/index.html

SMF (Service Management Framework) is a commercial OSGi framework developed by IBM. It is currently only available as a component in IBM's Workplace Client Technology Micro Edition, which at this point in time is at version 5.7. SMF implements the OSGi R3 specification.

3.4.6 Ubiserv

http://www.gatespacetelematics.com/products/ubiserv_osgi.shtml

Ubiserv is a commercial OSGi framework developed by Gatespace Telematics. The current version of Ubiserv is a complete certified OSGi R3 compliant service platform. Gatespace Telematics are the sponsors of the open source Knopflerfish framework, and Ubiserv can be viewed as the commercial version of Knopflerfish, as large parts of it are built on top of the open source alternative.

3.4.7 Jadabs

http://jadabs.berlios.de/index.html

Jadabs is a free OSGi framework developed at the Swiss Federal Institute of Technology, Zurich. It is built on top of the open source Knopflerfish framework, and is compliant to the OSGi R3 specification. Jadabs is specifically targeting small devices in a distributed environment.

3.4.8 Osxa

http://www.osxa.org/wiki/

Osxa is an open source OSGi framework developed by Rainer Alfoeldi and Harald Niesche. It implements most of the OSGi R4 specification, except for a few omissions to make the framework more lightweight. Specifically, these omissions make it possible to run the framework within an EJB container or in unsigned Java WebStart applications.

Name	OSGi spec.	Availability	Latest release
Knopflerfish 2.0	R4	Free (Open source)	05.04.06
Knopflerfish 1.3.4	R3	Free (Open source)	14.10.05
Oscar	R3	Free (Open source)	23.05.05
mBedded Server	R4	Commercial	2006
Equinox	R4	Free (Open source)	05.05.06
SMF	R3	Commercial	2004
Ubiserv	R3	Commercial	2005
Jadabs	R3	Free	22.06.05
Osxa	R4 (Incomplete)	Free (Open source)	22.03.06

Table 3.1: OSGi frameworks

All the frameworks are listed in Table 3.1, along with key information such as their OSGi specification version, whether they are available for free with open source, free with closed source or as commercial products, and the dates for their latest release. For UbiCollab, the most important features of an OSGi framework is whether it is under active development, so that any bugs and flaws gets removed quickly and because of the possibility of obtaining support, and whether it follows an open source model, so that the code can be used, or maybe modified, to assist with any problems.

3.5 OSGi in UbiCollab

OSGi satisfies the two non-functional platform requirements of being extendible and device operating system independent inherently, by providing a “general-purpose, secure, and managed Java framework that supports the development

of extensible and downloadable applications known as bundles." [20] This extendibility stems from OSGi's plug-in system and its bundle interdependency handling, and because it is based on Java, it is device operating system independent.

Additionally, OSGi has built-in mechanisms to check for present components, and is able to let the component developer decide what to do when a specific component is available or not. This way it also satisfies the flexibility and dynamicity requirement of letting different subsets of the platform be deployed on a system or device depending on the needs of the system's users or the capabilities of the device.

3.5.1 Benefits of OSGi for UbiCollab

Extendible – It has a built-in plug-in mechanism based on imports and exports.

Device operating system independent – It is Java and should therefore be able to run anywhere.

Dynamic – It is component-based and has service tracking mechanisms allowing for different configurations of the platform depending on the situation.

3.5.2 Possible issues with OSGi for UbiCollab

Memory heavy – It is Java and will therefore leave a relatively large memory footprint which can be problematic for smaller devices.

Technology specific – Since OSGi provides many benefits inherently to UbiCollab, it could be hard to port UbiCollab to another technology if OSGi becomes obsolete.

3.5.3 Deployment discussion

One of the most significant changes made to the platform in this version is the decision to spread the platform components among most of the participating devices in the environment. The fist version of UbiCollab did to some extent allow certain services to be located on different servers, but it was mostly a centralized solution, which did not allow platform components on client devices. Such a design has a few disadvantages which conflicts with the vision of a flexible platform for a dynamic environment. Fist, all applications have to relate to the same set of services. This makes it difficult to tailor the platform behavior to different settings and environments. Secondly, some services cannot be provided by the platform if no platform components exist on the client

devices. Positioning using GPS is one example of such a service. This service will have to be located where the GPS-receiver is. This is typically something that is carried around by the user, and the platform will therefore not be able to contact the receiver unless a platform component exists on the client device. Finally, a centralized solution has high traffic through the server and is prone to single-point-failures.

The current version of UbiCollab addresses these problems by spreading the platform components among the devices in the system. The usage of OSGi plays an important role in this new design, as it makes it easy to deploy platform components on the different devices. Because of this, it is possible to tailor the platform functionality to the capabilities and constraints of the different devices. A small device with limited storage and processing power could for example have just the most important platform components installed and rely on remote invocation for other tasks, whereas a more powerful device could run more services locally and thus boost performance. OSGi does also allow components to be easily updated without requiring a restart of the framework. This allows the new UbiCollab platform to automatically adapt to new environments by downloading and updating bundles at runtime. Finally, OSGi can allow bundles to be automatically pushed to - and installed on a large number of remote devices. This will make it easy for an administrator, such as Telenor, to automatically configure the platform for a large number of users. It should be noted that this last issue has not been properly tested with the current version and should be investigated further in later versions. Even though the new version of UbiCollab has a more distributed architecture than the previous one, it is not a completely distributed solution. The UbiCollab server has not been removed and does still host the collaboration service. This has been done because a completely distributed solution makes it a lot harder to handle things like synchronization of data. The risk of turning the server into a bottleneck should still be significantly reduced, though, as the work load on the server has been drastically cut.

3.5.4 Platform-internal communication

Two different means of communication are used to communicate between platform components. The preferred way is to use local OSGi calls. These are regular java invocations where the reference to the other object is provided by the OSGi framework. The reason why this is the preferred method is that OSGi is well suited to handle the dynamicity in the environment. By registering a ServiceTracker - object, one component can monitor another and get notifications if the other component is started, stopped or modified. Appropriate actions can then be taken based on this information. The problem with this technique is that it does not work across different OSGi - frameworks. UbiCollab does therefore use web service calls when communicating across OSGi frameworks. Further work on UbiCollab should investigate means of using OSGi calls across

different frameworks.

Figure 3.1 illustrates the platform-internal communication as well as communication between applications and the UbiCollab platform. It can be seen that services within the same OSGi framework communicates using local OSGi calls, whereas communication across OSGi frameworks is performed using web service calls. The latter does also apply for application-to-platform communication.

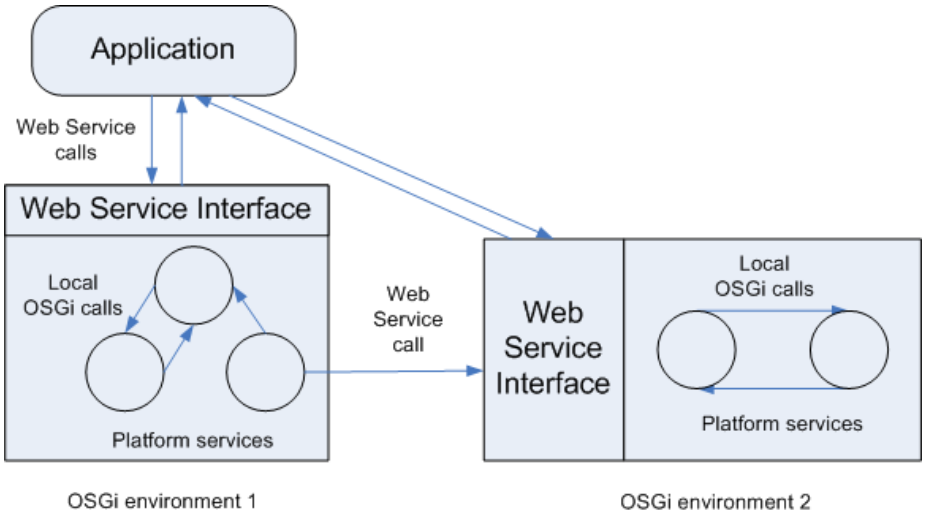

Figure 3.1: Internal platform communication

3.5.5 OSGi framework selection

When deciding the particular OSGi framework for the development and testing of UbiCollab, the most important aspects are that it is relatively free of bugs and missing specification implementations, and that it has a high degree of active development, signaling willingness and ability to assist with and fix any possible issues occurring throughout development.

The OSGi framework chosen to act as the underlying framework for UbiCollab during development is Knopflerfish 2.0. The main reason for this is Knopflerfish's frequent releases, and its thriving user community as a source of information in case of problems. Equinox would probably also have been a good choice as little actually separates them in terms of feature completeness and development activity.

Chapter 4

The UbiCollab platform

This chapter discusses the new architecture of UbiCollab, and presents the design and overall functionality of the service components chosen to form the platform's core services. Not all of these services are implemented fully according to the presented design in the first prototype of the new version of UbiCollab.

4.1 Introduction to the new platform

The most important aspects in designing a new platform for UbiCollab are platform extendibility, a non-functional requirement from the autumn report, and the high level flexibility of being usable and beneficial in a large set of different settings and scenarios [19]. As the requirements in the autumn report were extracted by finding commonalities between several different scenarios, adherence to these requirements would result in a platform with the desired flexibility.

4.1.1 Platform responsibilities

The APIs provided by the platform is based on functional blocks mainly derived from the platform requirements extracted in the autumn report [19]. These functional blocks are encompassed in logical components, forming the services of the platform. These services are:

Collaboration service

It provides applications with a way to manage Collaboration Instances. The service is not derived directly from any particular requirement, but it is needed on a higher level to support collaboration.

Discovery service

It gives applications the means to register and discover services and devices. The service is directly derived from the requirement Resource collection.

Identity manager service

It offers users privacy, and a way to maintain different user profiles depending on the setting. The service is directly derived from two requirements; Privacy and User profile management.

Context service

It presents applications with the possibility of managing and reading information about the context of different entities. The service is not a direct mapping from any of the requirements, but it is needed on a higher level as a flexible way of sharing implicit information.

Positioning service

It grants applications the means to position other users or devices. The service is a direct derivation of the requirement Positioning.

Location service

It provides applications with a way to map position coordinates to textual descriptions of physical locations. The service is directly derived from the requirement Location management.

Presence service

It gives applications the possibility of reading or updating the presence information about users. The service is a direct mapping of the requirement Presence management.

Data storage service

It offers applications the means to store and retrieve data persistently. The service is directly derived from the requirement Persistent data storage.

Two of the functional requirements from the autumn report are not covered directly by components in the UbiCollab platform. These are Security and Asynchronous communication support. Adequate security should be provided through any service that handles resources that may be valuable or private, such as the Data storage service and the Discovery service. Asynchronous communication support is provided by the platform in the form of the Data storage service, upon which applications can be created to simulate mailboxes, etc.

4.1.2 Service model

The old version of UbiCollab was focused on the benefits of the peer-to-peer model, where service providers and consumers communicate directly, after discovering each other through a central discovery mechanism. This model is kept in the new version of UbiCollab, with the platform service components being even more independent and having the ability to be deployed on both different and multiple hosts. This is made possible by having each platform component provide its service interface as a self-sufficient web service.

Additionally, certain services employ a three-tier mechanism where the service component itself only acts as a proxy between the consumer and the service plug-ins of the component. These plug-ins typically provide information from different technologies, which the service component then aggregates and returns to the consumer in a uniform fashion. These plug-ins are discovered and maintained by the service component through the Discovery service, and can therefore theoretically be deployed anywhere, for instance on a user's PDA or on a dedicated server for that plug-in.

4.2 Platform services

4.2.1 Collaboration service

The Collaboration service provides an API for applications to interact with Collaboration instances. This functionality includes the creation and removal of Collaboration instances, the adding and removal of services, persons and files to and from Collaboration instances, and functionality to retrieve information about a specific Collaboration instance and its services, persons and files.

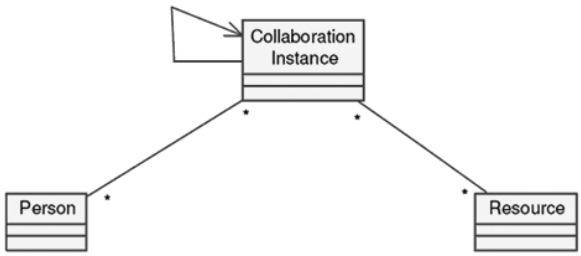

Figure 4.1: Collaboration instance [25]

A Collaboration instance is an abstract collection of persons and resources, for instance services and files, representing a real-world scenario such as a meeting

or a trip to the cinema. Schwarz [25] provides the formal definition of a Collaboration instance as "an entity which captures a real world activity, or context, of collaboration between people and the resources they use." Figure 4.1 shows a representation of a Collaboration instance, and its many-to-many relationship with Persons and Resources.

Collaboration instance data can be stored in many ways, for instance in a database or in a Subversion repository. Collaboration instances are stored on the central UbiCollab server, so the Collaboration service interacts directly with it. The Collaboration service is therefore dependent on the UbiCollab server running and that it can communicate with it.

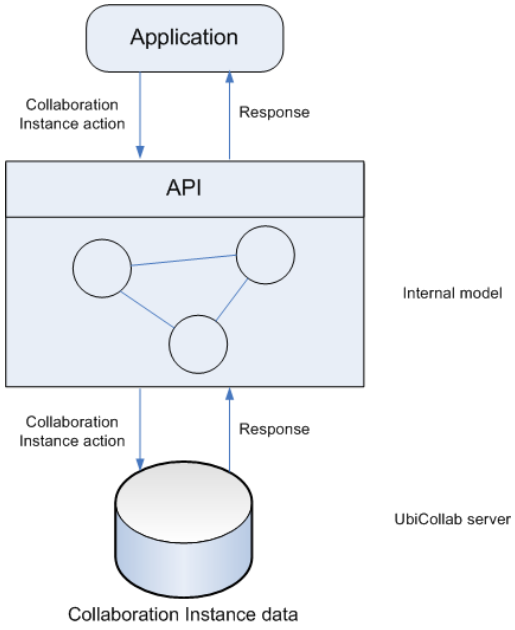

Figure 4.2: Collaboration service

In Figure 4.2, an application uses one of the methods of the Collaboration service API to perform an action on a Collaboration instance, for example listing all the persons associated with it or adding another service to it. The Collaboration service, in turn, issues the same action on its Collaboration instance data representation, and returns the response to the application. If the action was a request for information, that information is returned, and if the action was a manipulation of Collaboration instance data, a proper response code is returned.

Providing an API for getting information from and manipulating Collaboration instance data is important because it enables application developers to design

applications that take advantage of the collaborative benefits of the Collaboration instance model.

4.2.2 Discovery service

The Discovery service provides an API for the registration and discovery of services and devices. Some services and devices, such as those created according to the UPnP specification, are discovered automatically, while others have to be registered manually by the service provider.

In Figure 4.3, an application is using the Discovery service to obtain a handle to a particular service. The service could have been discovered by the Discovery service on any of its available discovery technologies, exemplified in the figure by UPnP, web services, Bluetooth or JXTA. In this figure, the internal model represents the searchable fields made available to other applications, either through the Discovery service's parsing of service description documents or through manual registry.

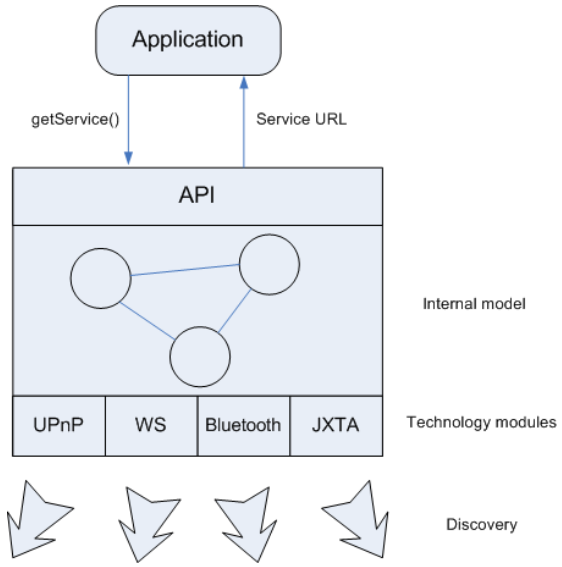

Figure 4.3: Discovery service

If it is desirable to support services on a new type of technology, a discovery module for that technology has to be plugged in, and the internal model has to be updated to handle the management of such services. The most important update would be to the parsing engine, to make it able to parse and register the service descriptions of the new technology type.

Services can be discovered automatically by the Discovery service or be explicitly added, depending on the technology the service is on. UPnP and JXTA services can for instance be discovered automatically, while web service-based services must be registered. This registry process can happen on start-up of the service, by scanning the service's RFID[1] tag with an RFID reader when the service is introduced to a collaborative setting, or manually by a user when none of the formerly mentioned methods are available.

4.2.3 Identity manager service

The Identity manager handles both a user's user profile, and the management of that user's virtual identities. Virtual identities are identities chosen for specific purposes, such as participating in collaboration without revealing personal data, or separating between work and home profiles.

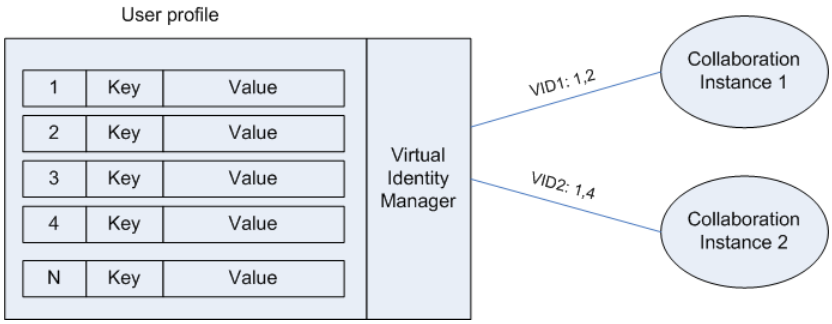

Figure 4.4: Identity manager service

The Virtual Identity Manager associates Virtual Identities with Collaboration instances, and with entries from the user profile. In Figure 4.4, Virtual Identity 1 is associated with Collaboration instance 1 and with entry 1 and 2 from the user profile, while Virtual Identity 2 is associated with Collaboration instance 2 and entry 1 and 4 from the user profile.

The possible keys and values in the user profile are not pre-defined, but are instead open for users to specify themselves. This way, different users can have different entries in their user profiles depending on their environment and setting, and applications can be created to look for certain keys and provide some specific functionality in case they exist.

[1] http://en.wikipedia.org/wiki/Rfid

4.2.4 Context service

Context information is in UbiCollab defined as any information that can be sensed or detected. The Context service will provide an API for the management of such context information. Because of the enormous variety in the information that can be stored in such a service, it will not be feasible to standardize specific context keywords and their data ranges. Instead, when an application is requesting context information, the Context service will return an XML file with all context key-value pairs, such as in Listing 4.1.

```
<context>
   <location>NTNU </location>
   <temperature>15C </temperature>
   ...
</context>
```

Listing 4.1

It will be the applications' responsibility to parse this XML file and extract the information it is capable of interpreting. For UbiCollab, contextual information will probably be of most value when related to a particular collaboration instance. For example retrieving the number of collaboration instance members currently in the meeting room, or whether your friends are awake yet and ready for some social activities.

In Figure 4.5, an application is requesting the current context of an entity, for instance another user. The Context service, in turn, queries all its information providers for information about that entity, and builds a Context XML that is returned to the application. The Context service keeps itself updated with the status of all possible information providers, so that these can be queried for information when necessary.

This way, the Context service acts as a layer on top of all information providers; both services, such as the Location service, and applications, such as different types of sensors. Instead of going to these directly, applications can use the Context service as a proxy to any kind of information that can be detected or deduced.

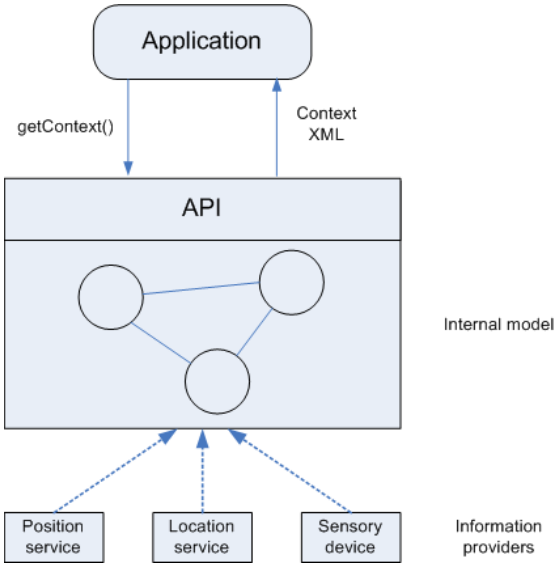

Figure 4.5: Context service

4.2.5 Positioning service

The Positioning service provides an API for retrieving the position of other entities. Because of a plug-in based system, the Positioning service is capable of retrieving position data, in the form of coordinates, from any type of coordinate-compatible positioning mechanism, as long as it has an interface conforming to the plug-in system. This could for example be GPS receivers, WLAN-positioning services, RFID tags on immovable objects, GSM-positioning services, and dedicated positioning services combining data from several sources. The Positioning service discovers such devices or services by querying the Discovery service for positioning plug-ins. When several positioning mechanisms are available, the service will have to decide on which one to use, or aggregate the result from several of the sources (e.g. calculate the average).

In Figure 4.6, an application is requesting the current position of an entity, for instance another person or a device such as a printer. The Positioning service issues the same request to its positioning plug-ins, and returns the result to the application.

The plug-in system works by having discoverable plug-in services for specific positioning technologies, and making sure that these plug-ins are standardized in the way they return position data. This way, the Positioning service uses the Discovery service to keep track of usable positioning plug-ins, and is able

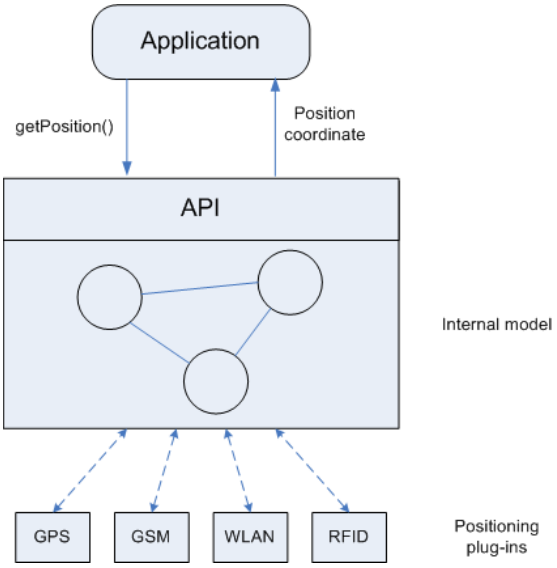

Figure 4.6: Positioning service

to issue the same type of query, and receive the result in the same format, to and from the different positioning plug-ins. Some of these plug-ins may be personal, such as a GPS positioning plug-in associated with the GPS receiver on a specific device, or public, such as a GSM positioning plug-in having access to the positions of several users.

4.2.6 Location service

The Location service provides an API for adding and removing locations, and for handling the mapping between a textual identifier of a location and a set of coordinates forming that location's physical boundary. The users are required to use another service or mechanism to first get the position of the entity they want to know the location of, but this design ensures that the Location service is not dependent on the presence of a Position service.

In Figure 4.7, an application is querying the Location service for the location associated with a particular position coordinate. The Location service matches that coordinate against its set of stored locations, and if any location contains that particular coordinate, its textual description gets returned to the application.

The coordinate sets consist of at least three coordinates, making out the vertices on the boundary of a polygon. The most important aspect of the Location service is then to determine whether a given coordinate point is contained by

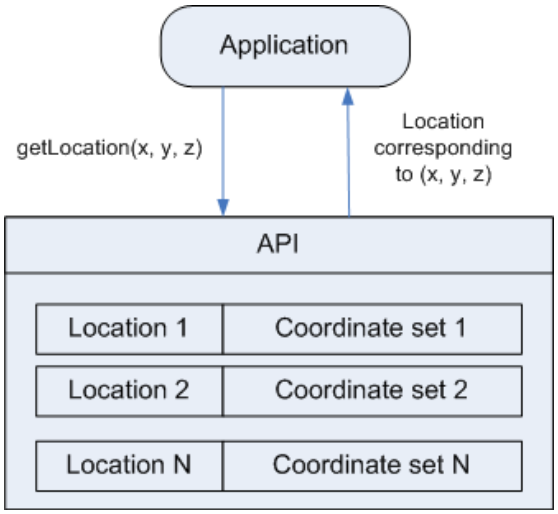

Figure 4.7: Location service

such a polygon, for instance whether a co-worker is in his office.

4.2.7 Presence service

The Presence service provides an API for managing the presence information of other persons. Presence information is in UbiCollab defined as a measure of how "close" two users are to each other. The presence value could for example be influenced by the physical distance between the users, whether the door between their offices is open or not, or by the existence of a video or voice link between them.

Presence information should be gathered by the system, with minimal need for user input or interaction. This means that the system should be sensor-based, to avoid having to force the users to specify their own presence state. Finding and developing such a system is something that needs much more research, and is an area which is not included in this work.

4.2.8 Data storage service

The Data storage service provides an API for storing, retrieving and deleting data on a persistent storage device. As the Data storage service employs a plug-in system, the data can be stored in many forms; for instance on a regular file system such as NTFS[2], in a database, or in versioning repositories such as

[2]http://en.wikipedia.org/wiki/NTFS

Subversion[3] or CVS[4]. If new types of storage systems are added, new technology modules have to be added, and the internal model of the Data storage service needs to be updated to be able to use the new storage system.

The core of the plug-in system in the Data storage service is illustrated in Figure 4.8, where an application asks the Data storage service to retrieve a certain file. The Data storage service then goes through its available data storage plug-ins, using storage device-specific communication between the technology modules and their respective storage devices. This can for example be Subversion or CVS repository access over HTTP, or regular FTP access to a file system.

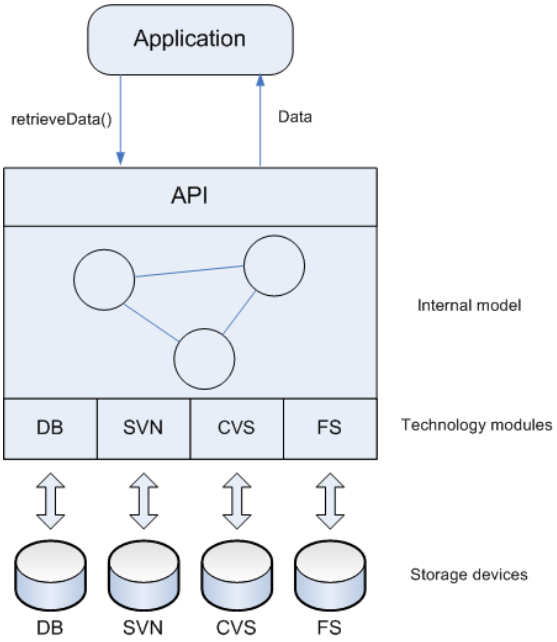

Figure 4.8: Data storage service

When the Data storage service receives a request for a file, it looks for that file in all its storage plug-ins, and returns it to the requestor if it finds a match. If more than one match is found, the Data storage service could return the latest version, or return all versions and let the requestor decide what file to choose.

[3]http://subversion.tigris.org/
[4]http://en.wikipedia.org/wiki/Concurrent_Versions_System

Chapter 5

UbiCollab implementation

This chapter discusses general implementation issues of UbiCollab, and then presents the implementation details of all service components that have been developed. These details include the full service API, and certain key points about the inner functionality. For the service components that are not fully developed, suggestions as to how this should be done are given.

5.1 Collaboration service

The Collaboration service provides access to UbiCollab's Collaboration instance data; both for reading and manipulating it. Currently, this data is stored in a database, but any other storage method would also be possible.

Because the data is stored in a database, the implementation of the Collaboration service is relatively trivial. All the functionality is related to the general database accesses; selects, inserts, updates, and deletes.

5.1.1 API

To create a Collaboration instance, the createCollaborationInstance() method, with the parameter username specifying the creator, and collabInstName specifying the Collaboration instance's name, is used. After such a creation, the Collaboration instance can either be removed by using the `removeCollaborationInstance()` method, or items can be added to it or removed from it by using the `add*` or `remove*` methods. These methods associate or disassociate the unique ID of a person, service or file with the specified Collaboration instance ID. Finally, the `get*` methods are available for retrieving sets of information from the system or from specified Collaboration instances. Currently, there are no security features implemented in the Collaboration service, so malicious applications are free to get access to other users' Collaboration in-

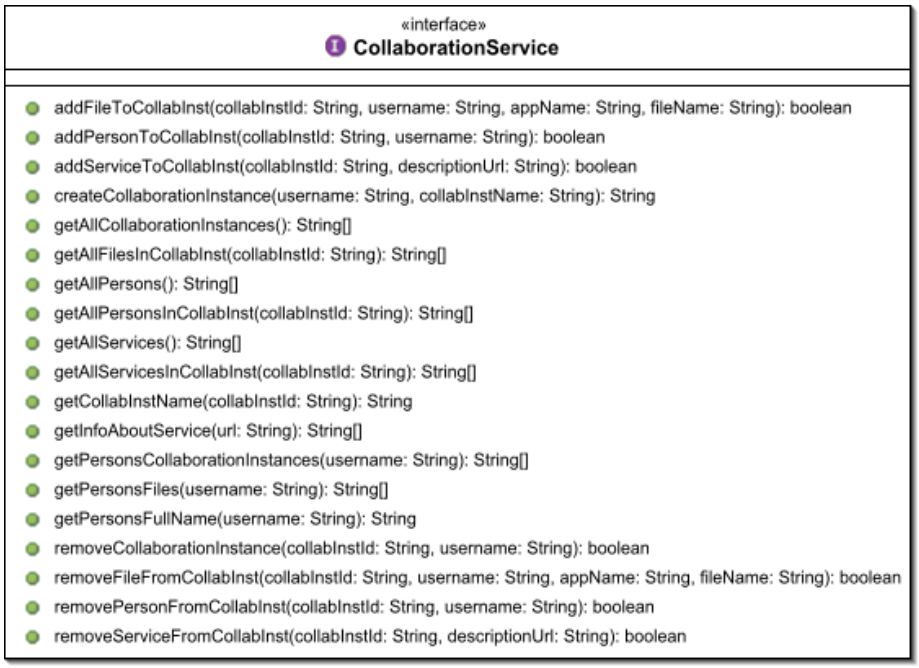

Figure 5.1: Collaboration service API

stances and their content by passing those users' usernames to the Collaboration service.

5.2 Discovery service

The Discovery service is responsible for keeping a repository of available services, and handing out handles to these when applications or other services need them. When registering services, the consumers of the Discovery service can either describe the full details of the service manually, or pass it a URL to the description XML of the service. If a URL is passed to the Discovery service, the Discovery service first checks if the URL is already registered as a service. If it is not, the Discovery service parses the XML into a DOM[1] tree, and traverses it to pick out certain details. The first objective is to establish what kind of service the document describes. Currently, the Discovery service supports the traversing of WSDL documents describing web services, and UPnP description documents. If the first tag of the document is <`wsdl`> the document describes a web service, and if the first tag is <`root`> with an attribute `xmlns` specifying the UPnP namespace `urn:schemas-upnp-org`, the document describes a UPnP

[1] http://en.wikipedia.org/wiki/Document_Object_Model

device. When the protocol type is established, the Discovery service traverses the DOM tree looking for specific nodes, such as the name and service URL for web services, and type and friendly name for UPnP devices. The service is then registered with these details in the Discovery service. Currently, the details are stored in a database residing on the same host as the Discovery service.

A possible issue with this parsing of description XMLs is that only a few key details are stored in the XML, and additional information that could be interesting to store, such as the service's owner or its fixed position in coordinates, is not possible to extract. Solutions to this would be to add the service manually using the method with the possibility of specifying all details, add a new method to the API of the Discovery service that takes in both an XML document to parse and additional details, or add a method that adds information to services that are already registered in the Discovery service.

Something that is lacking in this version of UbiCollab is a good classification of service types so that it is easy to find services of a specific type and automatically know what service it presents. Currently, the type field in the Discovery service is used to specify a namespace hierarchy such as `no.ubicollab.osgi.service.positioning` for Positioning services, and `no.ubicollab.osgi.service.positioning.gps` for the GPS positioning plug-ins. Having an ontology describing the services and their hierarchies would probably be a good way to classify the service types.

The Discovery service uses a plug-in system to cover discovery on different service technologies. Plug-ins are considered consumers of the service, as they automatically discover services on a specific technology, and then passes the description XML of that service to the Discovery service. Currently, the only plug-in that is added is a UPnP Discovery plug-in that is able to automatically discover and register UPnP devices.

5.2.1 UPnP Discovery plug-in

The UPnP Discovery plug-in works by utilizing the Siemens UPnP Control Point library for Java, which listens for UPnP SSDP packets on the network. Events are then triggered when such packets are sent on the network, and the data in the payload of the packets, most importantly the URL to the description XML of the service, is sent to event listeners in the UPnP Discovery plug-in. The UPnP Discovery plug-in then uses this URL to register the service in the Discovery service. Currently, the UPnP Discovery plug-in is implemented using the OSGi service tracker mechanism, making sure that it does not try to register services when the Discovery service is not running. The problem with this is that the UPnP Discovery plug-in then has to run in the same OSGi framework as the Discovery service. It is not hard to envision the benefits of running UPnP Discovery plug-ins in frameworks on different networks, all

reporting to the same Discovery service. Although the benefits of using the OSGi service tracker mechanism will disappear when plug-ins are run as web services, changes to the UPnP Discovery plug-in should be made so that it is able to exist in a different framework than the Discovery service. The only thing that has to be done for this to work is to include a proxy to the web service interface of the Discovery service in the UPnP Discovery plug-in, and update this according to what Discovery service is used. Additionally, the plug-ins should have a mechanism for temporarily storing URLs to services discovered while the Discovery service is unavailable, and register these when the Discovery service is available again. This could for instance be done by having the plug-ins check if the Discovery service replies to registry attempts, and if it is not, the plug-ins could store the URLs in a list and try to register them again regularly.

5.2.2 API

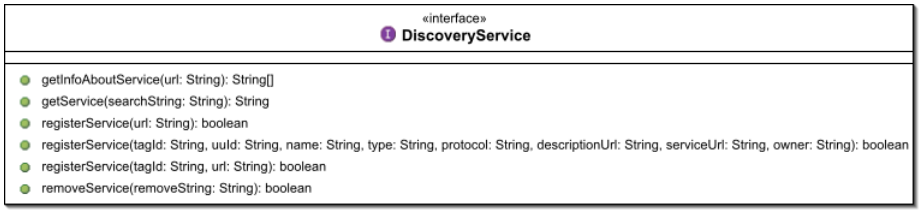

Figure 5.2: Discovery service API

There are currently three `registerService()` methods. One is for the situations where the only detail known is the URL to the description of the service. An example of this is the automatic discovery of UPnP devices through the UPnP Discovery plug-in. Then there is a similar method that also adds the parameter `tagId`. This is for services that are registered by reading the information on an RFID tag. This adds the possibility of searching for services based on the unique ID of the RFID tags. Finally, there is a method including parameters for all fields in the Discovery service. This is used to manually add specific information that can not be extracted from the description XML of a service.

Both the `getService()` method and the `removeService()` method uses a search string to specify which services to get or remove. The available flags for this search string are: `tagId:`, `uuId:`, `name:`, `type:`, `protocol:`, `descriptionUrl:` and `serviceUrl:`. Any number of these flags may be present, but they must come in the right order. An example of such a search string is `"name:My Service type:no.ubicollab.service"`, which would solely match against services with the name `"My Service"` and the type `"no.ubicollab.service"`.

The `getInfoAboutService()` method returns the full list of information registered about a particular service.

5.3 Pocket discovery service

This service is UbiCollab's solution to the chicken-and-egg problem of how to discover the Discovery Service, and is the only service that is mandatory in all platform instances. This means that applications and other platform components always can rely on having a Pocket Discovery Service running on the local host.

Even though this service is a mandatory service, it is not listed as one of the core services of the platform. This is because the Pocket Discovery Service does not introduce new functionality to the platform and simply acts as a proxy service, forwarding requests to the correct Discovery Service.

As the service is mandatory in all platform instances, it is of the essence that all devices running platform components can actually run the service. Therefore the service has been implemented with the focus of creating it as lightweight as possible. The current implementation of the Pocket Discovery Service does therefore not do any processing of its own, and instead forwards all requests to a Discovery Service. This does have one serious downside; it reintroduces the chicken-and-egg problem. How can the Pocket Discovery Service know where to find the Discovery Service? Currently, the Pocket Discovery Service retrieves the IP-address of the Discovery Service from a local configuration-file. This file can be set manually or updated automatically by either applications or other platform components via the service's interface. This does of course not solve the problem, but it makes it a lot more manageable. Instead of requiring all applications to keep track of where the Discovery Service is located, they can send their requests to the local Pocket Discovery Service and let it keep track of where to forward the requests. Consequently, the Pocket Discovery Service is the only service that will need to be notified if the address of a Discovery Service changes.

The interface to the Pocket Discovery Service contains the same methods as the Discovery Service plus one additional method. By providing the same methods as the Discovery Service, applications can invoke the methods on the Pocket Discovery Service as if they were invoking them on a Discovery Service.

The extra method found in the Pocket Discovery Service has the following signature: `boolean registerDiscoveryService(String url)`. By calling this method with a given URL, this URL will be set as the new address to the Discovery Service. The new URL is also written to the configuration file so that it is not lost in the case of a crash where the service has to restart. This method is accessible to both platform components and applications through the service's web service interface. An example of an application that could use this method

is an RFID reader application. If a user enters a room with a Discovery Service and the service is tagged with an RFID tag, such an application could be used to read the URL from the tag. The `registerDiscoveryService()` method can then be used to tell the platform to forward all requests to this new Discovery Service. Requests from other applications are automatically routed to this new service without the need for any modifications to the code or settings of the applications. The method can also be called by other platform components, for example if one Discovery Service is about to be replaced by another.

It should be noted that the Pocket Discovery Service is implemented with focus on making it as lightweight as possible so that it can run on all types of devices. More powerful devices could change the implementation so that more of the processing is done locally. Some could even run a fully functional Discovery Service as their local Pocket Discovery Service.

Currently, the Pocket Discovery Service can only keep track of the location of one Discovery Service at a time. Further work on the service should extend it to handle several Discovery Services. This would also make it possible to, at all times, forward the requests to the Discovery Service with the least load. It would also make the system more robust as the service could continue to operate even if one of the Discovery Services went offline.

Figure 5.3 shows the dataflow from the applications, via the Pocket Discovery Service, to the Discovery Service and back. As can be seen from the figure, the Pocket Discovery Service does not have a service repository on its own. Its job is simply to forward requests to a Discovery Service.

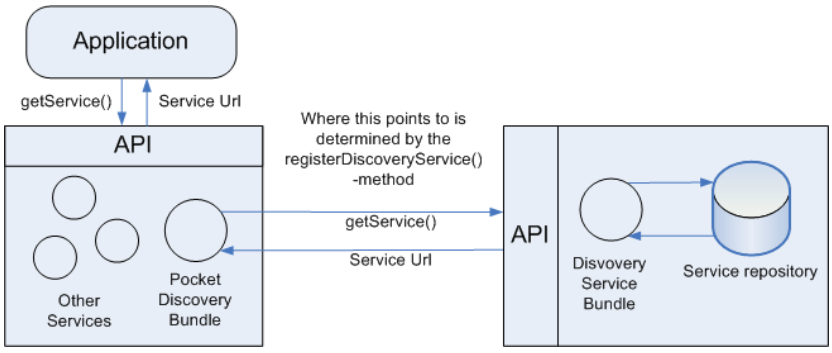

Figure 5.3: Pocket discovery service and Discovery service

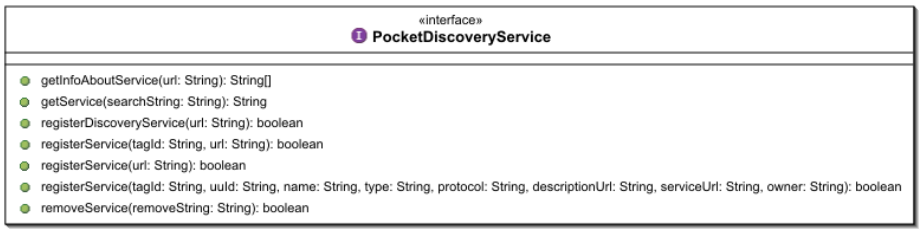

Figure 5.4: Pocket discovery service API

5.3.1 API

The Pocket Discovery Service contains the same the methods as the Discovery Service. As the Pocket Discovery Service just forwards the requests to a Discovery Service, the return values and parameters are also the same as was those in the Discovery Service. The `registerDiscoveryService()` method is the only method that is not found in the Discovery Service. This method instructs the Pocket Discovery Service on where to forward its requests.

5.4 Location service

The Location Service is responsible for the mapping between positions in coordinates and a location name. This is important as a name is a lot more meaningful to human readers than coordinates.

Internally, the location is represented by a `java.awt.Polygon` object. The polygon is wrapped by a `Location` class that, in addition to the polygon, contains the name of the location, the altitude, and methods for testing whether the location contains a given position. Two such methods exist. One that checks if altitude of the position is within a specified error margin of the location's altitude, and one that ignores altitude altogether. The `Polygon.contains()` method is used to test whether the given position is contained by the location. The `Location` class implements the `Serializable` interface and can therefore be serialized and written to persistent storage.

The location zone is two-dimensional, meaning that the location is registered with a single value for altitude, and consequently does not allow for any topological differences in altitude within the zone. For such zones, the average altitude of the area should therefore be used as the zone's altitude.

The service stores locations in an `ArrayList`, and provides methods for adding and removing locations from this list. Whenever the contents of the list is changed, the list is also serialized and saved to a file, so that the list is not lost

if the service have to perform a restart.

Two methods are provided for retrieving a location corresponding to a given position: One that takes altitude into consideration and one that ignores it. When these methods are called, the service will iterate through the list of locations, calling the `contains()` method on each location. When a location containing the position is found, the name of that location is returned to the caller. This means that only the name of the first location is returned even if the position is contained in several locations. Possible alternatives that should be considered in future versions include: returning all matching locations and returning the smallest, or most detailed, location in the case where one of the matching locations is fully contained in another.

On startup, the service loads the serialized list of locations from the persistent data storage, and deserializes it.

As the service is web service based, it cannot be automatically discovered by the Discovery Service. Consequently, the service has to register itself with the Discovery Service in order to become discoverable to applications and other platform components. Therefore, the service searches its local OSGi environment for a Pocket Discovery Service upon startup. If found, the Location Service invokes a local OSGi call to the Pocket Discovery Service which uses its associated Discovery service to register the Location Service in the service repository. As the Pocket Discovery Service is mandatory in all platform instances, it should run in the same OSGi environment. However, there might be situations where the Pocket Discovery Service is temporarily unavailable. To make the system more robust, the Location Service uses a backup plan in these cases, and instead registers itself with a remote Discovery Service. The address to this service is retrieved from the Location Service's configuration file. These two services are not found in the same OSGi environment, and the registration is therefore performed using a web-service call.

5.4.1 API

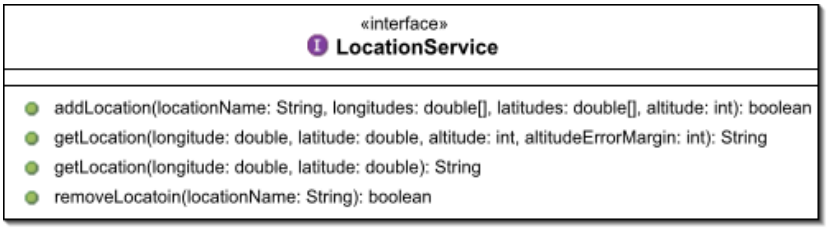

Figure 5.5: Location service API

The Location Service contains methods for adding and removing locations from the internal list. The two `getLocation()` methods returns the name of the first location that contains the given position. The difference between them is that one ignores altitude whereas the other requires the altitude of the given position to be within `altitudeErrorMargin` of the registered altitude of the location.

5.5 Positioning service

The Positioning Service is responsible for finding the current position of users and resources.

Positioning can be performed using several different technologies [12]. GPS, GSM and WLAN are examples of such technologies that exist today. In the future even more options will become available. The European Galileo satellite navigation system[2] is for example scheduled to be fully operational by 2010. No single system can be used effectively in all scenarios, and none are guaranteed to be around forever. Therefore, this service has been developed as a plug-in-based service. This has the advantage that the service can be easily extended with new plug-ins as new positioning techniques appear, and support for outdated technologies can be terminated by simply deleting the plug-in. No change to the source code of the Positioning Service is required.

Currently, two plug-ins are implemented; a GPS plug-in and a GSM plug-in. The GPS plug-in is fully implemented, and retrieves positions by connecting to a GPS receiver. The GSM plug-in, on the other hand, is currently a stub implementation and does only return hard coded coordinates. The plug-ins are described in detail below.

The service provides only one method for retrieving the position of a user or resource: `double[] getPosition(String username)`. When this method is called, the service asks all its registered plug-ins for the position of the user. This makes it possible for the Positioning Service to compare the data from several sources and pick the most accurate one. The current version has a hard coded order of prioritization, saying that GPS data should be used instead of GSM data when both are available, and that GSM data should be returned before data from other types of plug-ins. However, later version should have more intelligent behavior, and could for example combine the data with contextual information so that WLAN positioning is used instead of GPS when the user is indoors.

There is a many-to-many relationship between Positioning Services and its plug-ins, meaning that in addition to a Position Service having more than one plug-in,

[2] http://en.wikipedia.org/wiki/Galileo_positioning_system

the same plug-in can be "plugged into" several Positioning Services. This is advantageous, as it prevents a monopolistic situation where a plug-in is exclusive to one particular Positioning Service. If that was the case, the service provided by the plug-in would become unavailable if the Positioning Service owning it went offline.

As OSGi is a well suited environment for a plug-in based system, a natural choice would be to let the Positioning Services and plug-ins interact using local OSGi calls. This is, however, not possible in this scenario since the plug-ins are not necessarily found in the same OSGi environment as the Positioning Service. A GPS plug-in would for example typically be found in the OSGi environment of the mobile device where the GPS-receiver is found, whereas the Positioning Service could be located on a completely different node in the network. The Positioning Service does instead interact with plug-ins using web service calls.

One of the overall design goals of UbiCollab is to make it capable of operating in a highly dynamic environment. Consequently, it is not known at design time what kind of plug-ins might be introduced to the system. This poses a serious challenge: The service will need proxy classes to be able to connect to and invoke methods on the plug-ins, but these cannot be created at design time as it is not known what kind of plug-ins will be found in the environment. UbiCollab solves this problem by creating proxy classes to the plug-ins at runtime. Upon startup, the service searches the Discovery Service for positioning plug-ins using the search string: `"type:no.ubicollab.osgi.service.positioning.* protocol:ws"` The trailing `'*'` in the type definition is a wildcard and does therefore match all types of positioning plug-ins. The service then parses the WSDL description of each plug-in and checks for a method with the signature `double[] getPosition(String)`. If such a method is found, a proxy is created and the plug-in is added to the plug-in list. In this way, any service exposing this method could be used as a plug-in. It should be noted, however, that the current implementation does not allow Positioning Services to be used as plug-ins, as it could cause the two Positioning Services to invoke methods on each other in a never ending loop. This is prevented by putting the Positioning Service in a different namespace. This service will therefore not be returned by the Discovery Service when using the search string above.

In addition to doing the plug-in search on startup, the references are automatically updated every 35 minutes. Applications can also force an update by calling the method `updatePluginReferences()`.

Figure 5.6 illustrates, slightly simplified, the interaction between an application and the involved platform components in a typical scenario where the application is requesting the position of a user. The upper half of the diagram shows how the Positioning Service queries the Discovery Service for compatible plug-ins and dynamically creates proxy classes from the returned URLs. The bottom half demonstrates how the Positioning Service, when receiving a "get position"

request, sequentially queries all its registered plug-ins until the position has been found, or it has been determined that the position is not found in any of the plug-ins.

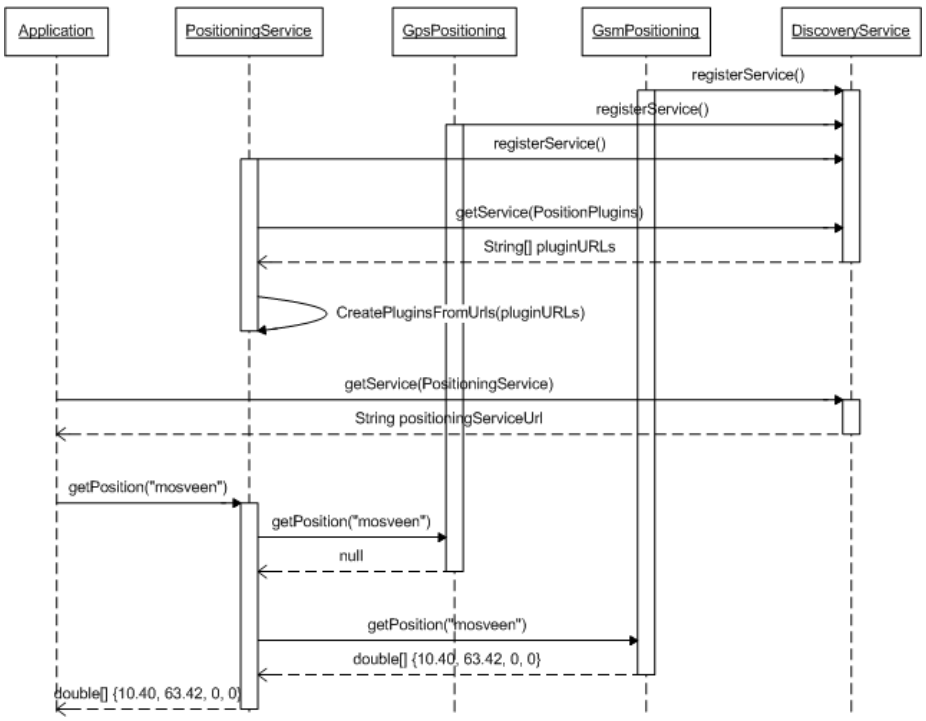

Figure 5.6: Positioning interaction

This process of iterating through the list of plug-ins until the position has been found, works fine in most cases, however, it is an inefficient solution in the cases where the number of plug-ins is high and each plug-in only can retrieve the position of a few users. A high number of GPS plug-ins would for example create such a situation, as the GPS plug-in can only return the position of its owner. In these cases, the current solution would unnecessarily invoke the `getPosition()` method on several plug-ins that do not have the position of that particular user. A better solution that should be considered in future versions is to include the username of the owner as part of the plug-in WSDL description. The Positioning Service would then know immediately which GPS plug-in to ask for the position of the user. Another alternative is for the Positioning Service to quire the Discovery Service for a plug-in that has the position of that particular user.

5.5.1 API

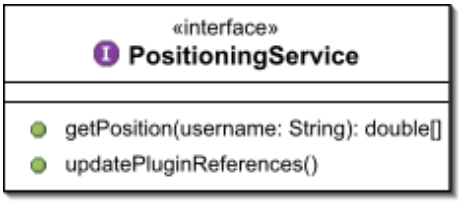

Figure 5.7: Positioning service API

The Positioning Service's interface contains only two methods. The `double[] getPosition(String username)` method returns the position of the user with the specified username. In future versions it should also be possible to retrieve the position of resources by passing the resources' URL as the "username", but there is currently no way of registering a resource's position. The position is returned as a double array of length four, containing the longitude, latitude, altitude and timestamp of the fix[3], in that order. The second method: `updatePluginReference()` is used to force the service to update the list of plug-in references and re-create proxy-classes to these.

5.5.2 GPS Positioning plug-in

The GPS Positioning plug-in is a fully implemented plug-in to the Positioning Service, and works as a platform interface to a GPS receiver. The service exposes only one method: `double[] getPosition(String username)`. The signature of this method is essential as this is what makes the service a compatible plug-in to the Positioning Service. When the method is called, the service opens a connection to the GPS-receiver through a COM port. The name of the COM port is retrieved from the service's configuration file. The service then reads and parses the data from the GPS-receiver. Finally, the longitude, latitude, altitude and timestamp is extracted and returned as a double array.

In order to manage COM ports from Java, the service is dependent on having the Java Communications API installed on the device. This is not much of a problem, though, as implementations of this API is available (either from Sun or third-party vendors) for all common platforms.

The plug-in has been tested with the ITerNet PS-3100 Bluetooth GPS[4], on Windows XP and Windows Mobile 2003, but it should also work with other receivers and on other platforms as long as it can be accessed through a COM port. The

[3] http://en.wikipedia.org/wiki/Position_fixing
[4] http://www.iternet.com.tw/b-gps/ps3100-f.htm

plug-in also includes its own implementation of a parser for the $(GP)RMC command of the NMEA-0183 GPS standard[5]. This protocol is used by many GPS receivers; however, the parser can easily be swapped with another one in order to support receivers using other protocols.

The plug-in exposes its interface as a web service and can therefore be easily accessed by other applications. This is, however, strongly discouraged. Applications should instead use the Positioning Service as this will be able to retrieve position data from several sources and can therefore return the result from the most accurate source, at all times. In addition, the GPS plug-in will normally only be able to return the position of its owner. The Positioning Service, on the other hand, has references to several plug-ins and consequently has a better chance of finding the position in one of them.

In the same way as the Location Service, this plug-in will register itself in the local Pocket Discovery Service on startup. It is also capable of registering itself in a remote Discovery Service in the rare cases where the Pocket Discovery Service is unavailable.

API

Figure 5.8: Positioning plugin service API

The only method exposed by the GPS Positioning plug-in is the `double[]getPosition(String username)` method. When called, the plug-in will check if the username passed as a parameter is the username of the user who owns the connected GPS-receiver. If so, the service will retrieve the position from the GPS-receiver and return it as a double array with the data: longitude, latitude, altitude and time of fix. Otherwise, the service does not know the position of the user and will therefore return null.

5.5.3 GSM Positioning plug-in

The GSM Positioning plug-in is supposed to retrieve positioning data from the GSM-network. Many telecom providers, including Telenor, provide such ser-

[5]http://www.nmea.org/pub/0183/

vices in their networks. The GSM Positioning plug-in will access these services and thus act as a platform interface to this positioning technique. The current implementation is, however, just a stub implementation and does only return hardcoded positions for a set of manually added users.

API

Figure 5.9: Positioning plugin service API

It is important that all plug-ins return the same type of data. This plug-in does therefore return the same information as the GPS plug-in. The only difference is that the data is retrieved from the GSM network instead of a GPS receiver.

5.6 Data storage service

The Data storage service provides applications and other services with the possibility of persistently store data. While the service has a plug-in based design, which is discussed later, the current implementation only stores data on the local disk storage. To do this, it uses regular Java file I/O, which is platform independent.

On start-up, the Data storage service registers itself in the Discovery service through the use of a Pocket Discovery service. Since the Data storage service exposes a web service interface, data is passed to and from the Data storage service, through SOAP, as arrays of bytes. Because of the default size of the Java Virtual Machine's memory heap, this approach currently limits the size of the transferred data to about 2 MB. Larger files can be transferred if the memory heap is extended by adding the flags "-Xms*size in bytes*" and "-Xmx*size in bytes*" when starting Knopflerfish (e.g., `"java -jar -Xms128m -Xmx512m framework.jar"`). Files are currently stored in the directory `user.home/UbiDataStorage` where `user.home` is a Java property for the platform specific home directory of the user. Inside this directory, files are stored in the format `username/appname/filename`, meaning that every user gets his or hers own directory, with application directories underneath, and files within these again.

5.6.1 Plug-in design

The plug-in system design of the Data storage service is a result of a combination of two other designs that both had important drawbacks. The first one was a design where each data storage system, such as a database, would have to be equipped with its own Data storage service, and where these data storage systems all would be registered in the Discovery service. Then applications would query the Discovery service for a suitable Data storage service and use that directly. The drawback of this design was that it would require the existence of a Data storage service on each storage system, severely slowing down the deployment of usable storage systems. The second one was a design where the Data storage service would be a required component in every UbiCollab platform, and where that service would keep track of data storage systems as plug-ins. In such a design, the Data storage service would not need to be installed on all storage systems, as long as the storage systems themselves had an API for accessing their content. But on the negative side, it would mean one additional required component in the platform, which is something that should be avoided.

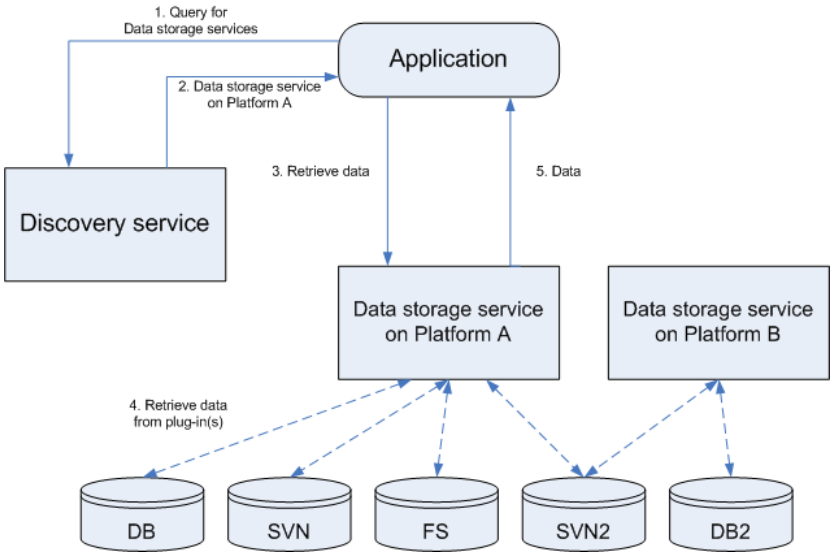

Figure 5.10: Data storage service overview

The final design keeps the idea of standard data storage plug-ins from the second design, without the need of installing prepared UbiCollab-specific interfaces on them, and keeps the idea of not being dependent on a required platform Data storage service from the first design. It achieves this by having both the Data storage services and the storage plug-ins discoverable in the Discovery service. As shown in Figure 5.10, an application would then choose a suitable Data stor-

age service, which in turn is associated with a set of data storage plug-ins. If the application user does not at first find the file he or she is looking for, he or she could try another Data storage service that perhaps would have access to other data storage plug-ins, as in Figure 5.10 represented by the Data storage service on Platform B and its DB2 plug-in. The current implementation of using the local storage should in this setting be extracted and redesigned as a file system plug-in.

5.6.2 API

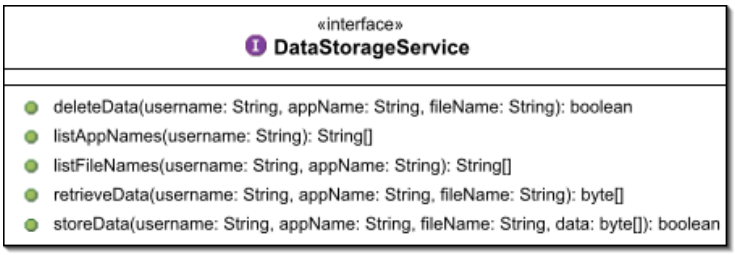

Figure 5.11: Data storage service API

The exposed methods for the Data storage service are very straightforward and easy to understand. They all use the naming scheme of `username/appname/filename` to identify the needed objects. The `list*` methods shorten this naming scheme to list all entries within one directory, while `storeData()`, `retrieveData()` and `deleteData()` uses the full naming scheme to specify a particular file. The file is stored and retrieved as an array of bytes, passed through SOAP, so it does not matter what platform the consumer application is running on or which programming language it is written in. Currently, there are no security features implemented in the Data storage service, so malicious applications are free to get access to other users' files by passing those users' usernames to the Data storage service.

5.7 Context service

The Context service is currently not implemented, but here follows suggestions and details on how this could be done.

To function as intended and be able to provide a wide set of context information, the Context service must be able to find and maintain a list of services and devices it can get information from. This means having a way to classify such items, for instance by having them in particular namespaces, specified by the

item's type. In the current namespace system, it would be natural to just use a regular namespace depending on the item, such as `no.ubicollab.osgi.service.positioning` for Positioning services, and register this namespace in the Context service. This system would be very fragile over time, and it stresses the importance of designing an ontology for the services and devices in UbiCollab. In this ontology, it would be possible to specify that a certain service or device also was a context information provider, which the Context service could use to build its Context XML.

There are two main alternatives to consider when developing the Context service. The first is a Context service that queries all its information providers when it receives a request for context information, and then builds the Context XML on the fly. The second is a Context service that queries its information providers and builds the Context XML in regular intervals, and serves the Context XML directly when it receives a request for context information.

The advantages of the first approach is that it always provides current context information, and that it does not need to store any data for longer than it takes to build and send the specific Context XML. The disadvantage of this approach is that, depending on how many information providers the Context service knows of, the process of contacting and receiving information from all of them could take a long time. The advantage of the second approach is that applications requesting context information will receive the specific Context XML straight away, without the Context service having to query all its information providers. The disadvantages of this approach is that the context information contained in the Context XML may be off by as much time as the Context service uses for its information provider query intervals, and that the Context service needs the ability to store several Context XML documents. The number of Context XML documents the Context service needs to store is dependent on how many items in the system have a specific context (e.g., persons and collaboration instances).

5.7.1 API

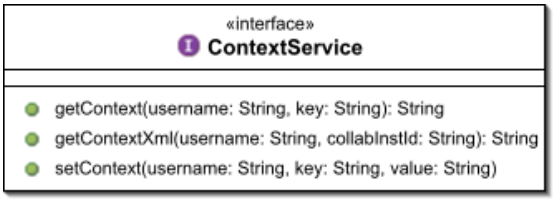

Figure 5.12: Context service API

Since the service is not developed, the service API is very basic at this point

in time. There are methods for getting and setting a context entry, and for retrieving the full Context XML.

5.8 Identity manager service

The Identity manager service is currently not implemented, but here follows suggestions and details on how this could be done.

The Identity manager service provides two areas of functionality; user profile management and Virtual identity management. Since these intertwine, and user profile entries are part of the Virtual identity management, the two areas of functionality are combined into the same service.

Functionality for the user profile management is relatively trivial, as the service only needs to keep a table containing the user profile entries, and manage an interface letting applications add new entries, or read or modify existing ones.

The Virtual identity management is somewhat similar, as it only needs functionality for creating new Virtual identities, and ways of associating unique IDs to user profile entries and Collaboration instances to these Virtual identities.

5.8.1 API

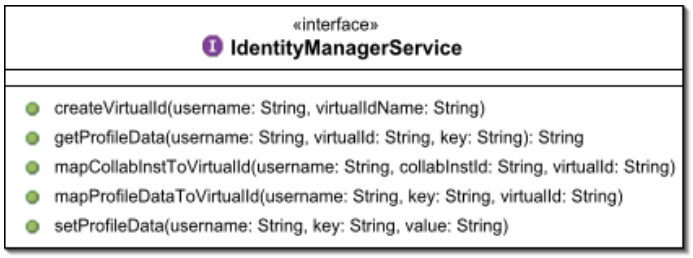

Figure 5.13: Identity manager service API

Since the service is not developed, the service API is very basic at this point in time. There are methods for setting and getting entries in the user profile, as well as for creating Virtual identities and mapping Collaboration instances and user profile entries to these.

5.9 Presence service

In this work, no research has gone into how to best represent presence information in UbiCollab, as this is out of the scope of the current task. Presence management is in itself a large research area, and more research will have to be dedicated on how to handle presence in UbiCollab. There will therefore be no discussion on the implementation of the Presence service, nor will this work present the service's API, as this would be too abstract at this point in time.

Chapter 6

Platform demonstration

This chapter introduces the testbed used for testing and evaluating the platform during and after development. It also presents each application and service made with the intent of testing different combinations of platform services.

6.1 Testbed overview

The first version of UbiCollab came with an application called UbiClient [11]. The purpose of this application was to demonstrate the use of UbiCollab to support collaboration in a meeting room scenario. This was a relatively large application which tested most of the functionality of the UbiCollab platform. The test-bed used to demonstrate the new version of UbiCollab, on the other hand, follows a different approach. Instead of using one large application to demonstrate and evaluate the platform, several small applications have been created. Each application is responsible for testing a small part of the core functionality of the platform. This approach has been chosen based on research by Edwards et al. which suggest that several small, lightweight applications that test core functionality are more effective in the early phases of platform evaluation [7].

There is also another factor that prevents the use of only one application to demonstrate the platform functionality. One of the design goals of the new version has been to make the platform more flexible so that it can be used in more environments and scenarios. It would be difficult and not realistic to include support for several such scenarios in the same application.

The platform bundles have been deployed in the testbed as illustrated in Figure 6.1. In this test bed, one of the laptops acts as the UbiCollab server, and thus runs the Collaboration Service. It should also be noted that the Pocket Discovery Service is found on all devices as this is a mandatory service in all platform instances. Some other services are also found on more than one device, e.g., the

Location Service. This is done to illustrate and test that several instances of the platform services can be found in the system at the same time.

A couple of other resources are also part of the testbed. The two light devices connected to one of the laptops are UPnP resources and are included to test resource collection and usage of resources.

In addition, a GPS receiver and an RFID reader are connected to the PDA. The GPS receiver is used by the GPS plug-in bundle to receive coordinates from positioning satellites, whereas the RFID reader is used to facilitate RFID discovery of services and resources.

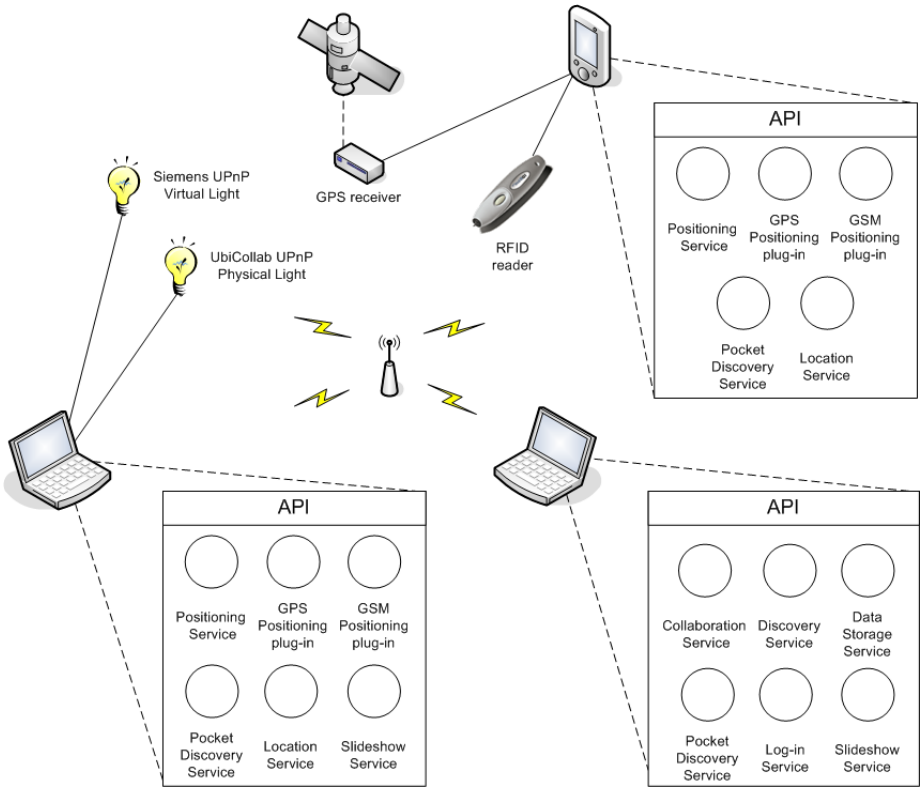

Figure 6.1: Testbed overview

6.2 Demo applications

6.2.1 UbiCollaborator

UbiCollaborator is a Collaboration instance management tool, of which the GUI is shown in Figure 6.2. The program is created as a standard Windows .NET application and works by communicating with UbiCollab's Collaboration Service. On startup, the program will show the current collaboration instances of the user. It will also provide means for creating new collaboration instances and deleting existing ones. In addition, the program can be used to add users, services and files to the Collaboration instances.

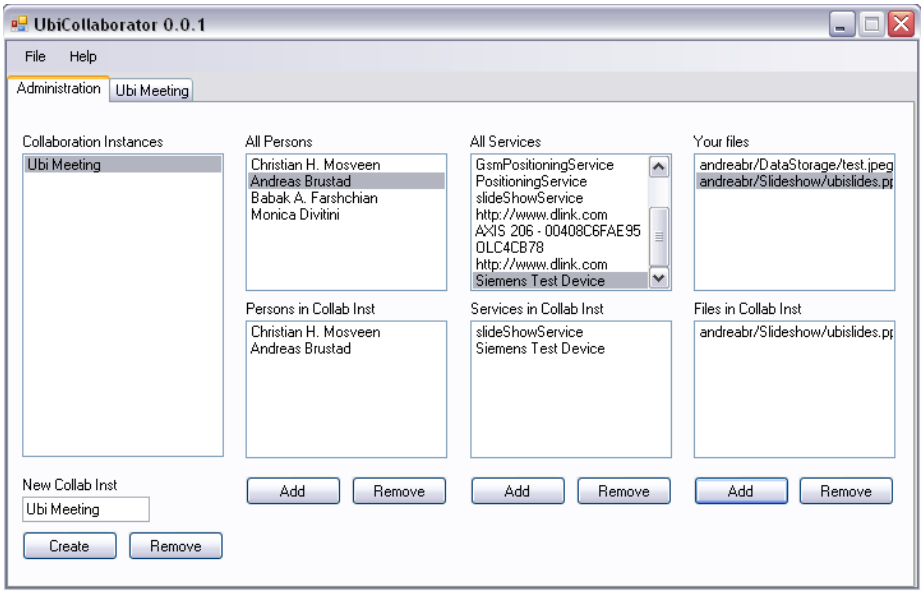

Figure 6.2: UbiCollaborator

6.2.2 Service registry

The purpose of this application, shown in Figure 6.3, is to register services and resources in the Discovery Service by reading the service's description URL from any attached RFID tags and sending it to the Discovery Service. The application can also read URLs pointing to Discovery services from RFID tags and instruct the platform to use the Discovery service at that URL as the primary Discovery service. The application reads RFID tags by opening a Bluetooth connection to an IDBlue Bluetooth RFID pen. The data is then read by the pen and returned to the application.

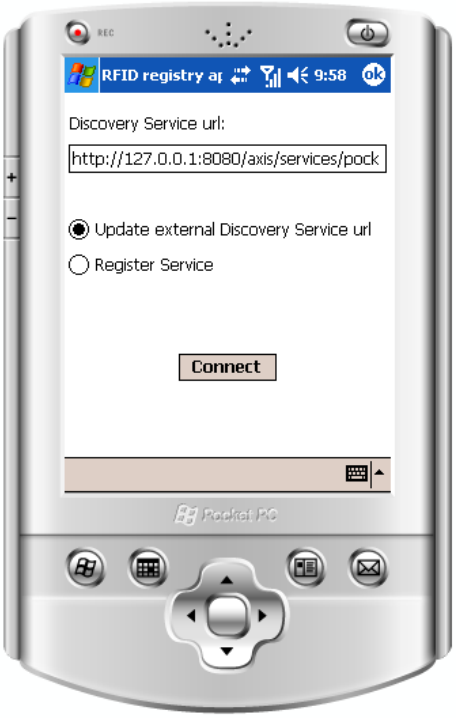

Figure 6.3: Service registry

6.2.3 UPnP Light control

The UPnP Light control is a remote control for UPnP enabled light devices conforming to the UPnP BinaryLight:1 standard. The application uses the platform's Pocket Discovery Service to search for compatible lights. Once detected, the application can invoke operations on them via the UPnP protocol. These operations include turning the light on and off, and requesting the status of the light device. The application is shown in Figure 6.4.

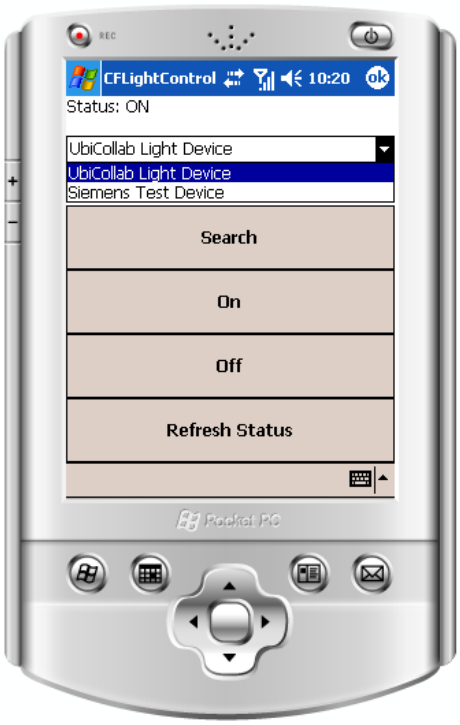

Figure 6.4: UPnP Light Control

6.2.4 Locator

The purpose of the Locator program, which is shown in Figure 6.5, is to test the positioning functionality of the UbiCollab platform. To do so, this Pocket PC application, communicates with both a Collaboration Service, a Positioning Service and a Location Service. The Collaboration Service is used to find all the other users in the current user's collaboration instances. The position of these users are then retrieved from the Positioning Service and displayed. Finally, the application sends these coordinates to the Location Service and retrieves the corresponding locations.

Figure 6.5: Locator

6.2.5 Positioning Service Map

As shown in Figure 6.6, the Positioning Service Map is a web-based application that displays position information retrieved from UbiCollab's Positioning Service on top of Google's Google Maps service[1]. This application is created to demonstrate how UbiCollab's services can be combined with 3rd party services to produce even more powerful results. As this is a web-based application, it does also demonstrate UbiCollab's platform and programming language independence.

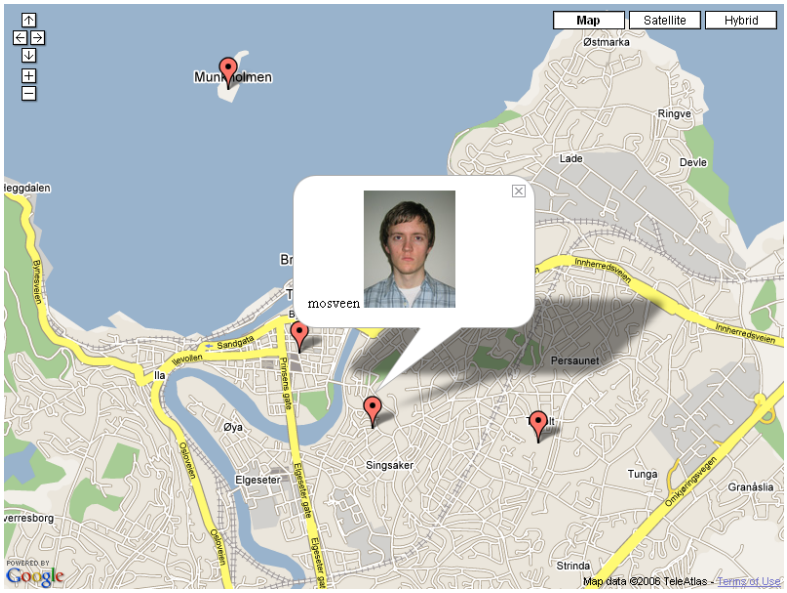

Figure 6.6: Positioning Service Map

[1] http://maps.google.com/

6.2.6 Slideshow Control

The Slideshow Control is created with the purpose of displaying and controlling Power Point slideshows on local or remote devices. The Slideshow Control consists of two components: A Pocket PC application and a UbiCollab Slideshow Service. The service component is capable of retrieving a specified PowerPoint file from the Data storage Service and open it in Microsoft PowerPoint. Once opened, the service can move forwards and backwards in the slideshow in addition to stopping the slideshow. As shown in Figure 6.7, the application component will connect to UbiCollab's Collaboration Service and retrieve all the files available in the current user's Collaboration instances. In addition, the application will use the platform's Discovery Service to search for available Slideshow Services, shown in Figure 6.8. The retrieved files can then be sent to, and displayed by the Slideshow Service, and controlled through the GUI presented in Figure 6.9. The application is also capable of synchronizing two or more Slideshow Services so that they display the same slides.

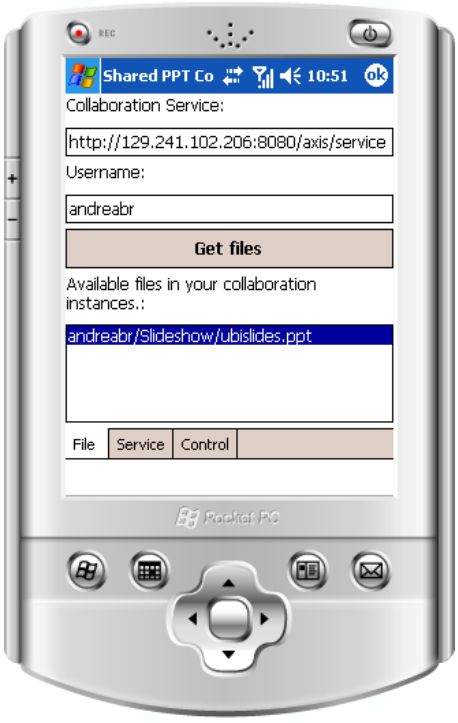

Figure 6.7: Slideshow Control: File

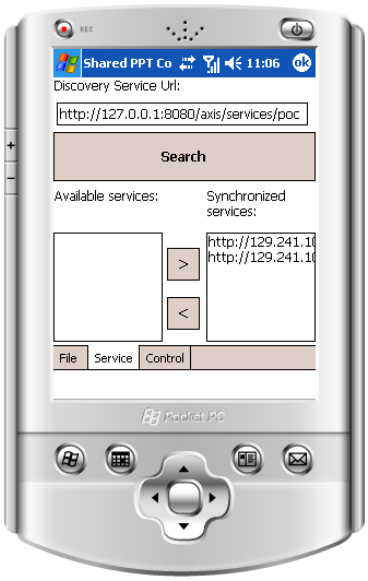

Figure 6.8: Slideshow Control: Service

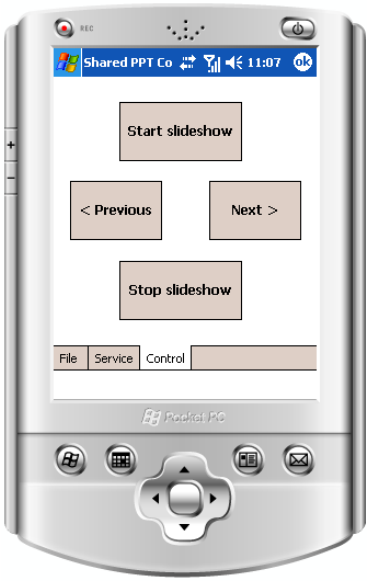

Figure 6.9: Slideshow Control: Control

6.2.7 Data storage Service Tester

The Data Storage Service Tester is, as the name suggests, designed to test the functionality of the Data Storage Service. The application is a standard .NET Windows application that allows the user to upload files to the Data Storage Service, download files that are owned by the user and list all of his or her files. The application is shown in Figure 6.10.

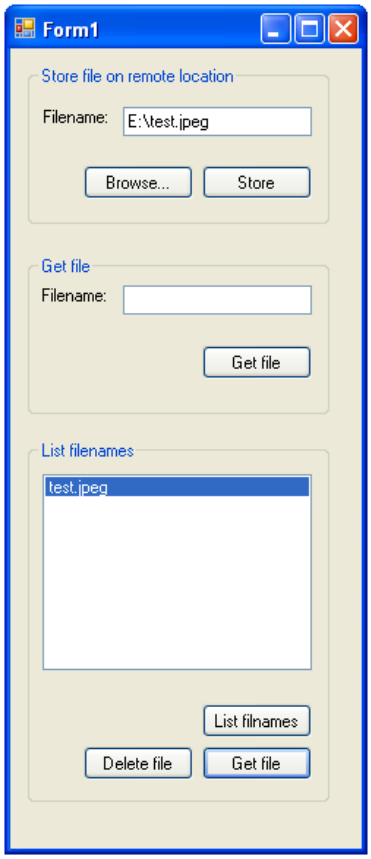

Figure 6.10: Data storage Service Tester

6.2.8 Login Service

The Login Service is designed to allow a user to log in to other devices in his or her vicinity, such as public displays. Similar to the Slideshow Control, The Login Service consists of two parts: A Pocket PC .Net Compact Framework application

and a UbiCollab service named Login Service. The .NET application is intended to run on a badge-like device, i.e., a device without display that only stores user credentials, whereas the Login Service should be installed in the UbiCollab platform on the public display. In order to log in to the public display, the user can simply scan the attached RFID tag. The application will then send the badge's user credentials to the Login Service at the URL read from the tag. If the Login Service receives a valid username and password combination, the display will show user data such as the username and the user's current collaboration instances, as shown in Figure 6.11.

Figure 6.11: Login Service

6.2.9 RFID Tag Writer

The RFID Tag Writer is a Pocket Pc application for reading, writing and clearing the data stored on an RFID-tag. This is accomplished by utilizing the IDBlue Bluetooth RFID pen. As this application does not in any way use the UbiCollab platform, it is only indirectly part of the testbed. The RFID tags and the data this application stores on them, on the other hand, plays an important role in the testbed.

6.3 Platform tests

6.3.1 UbiCollaborator

Collaboration instances form the basis for all collaboration support in the UbiCollab platform. Imagine for example a software engineer preparing for a meeting where his or her team will discuss the latest architectural changes to a program. Before the meeting can take place, he or she would have to invite the

other persons to the meeting. He or she might also want to let them know what they will be discussing, and thus include the meeting slides or a document to the invitation. Finally, he or she could check the meeting room to make sure all the necessary equipment such as a projector and a coffee machine is there.

Together these form a Collaboration instance in the UbiCollab platform. Creating such an instance is therefore usually the first step when using UbiCollab as an aid for collaboration. All Collaboration instance management tasks are handled by UbiCollab's Collaboration Service. Making sure this service work as intended is therefore extremely important.

The UbiCollaborator application is a Collaboration instance management tool designed to test this functionality of the Collaboration Service. With this tool, Collaboration instances can be created, deleted and modified, users can be added and removed, and services and resources can be included and deleted from the Collaboration instance.

In order to pass the test, the Collaboration should successfully process all of these requests from the UbiCollaborator application.

6.3.2 Service registry

Easy access to available services and resources are an essential part of a ubiquitous environment and important part of the UbiCollab platform. The above scenario can be used to illustrate this. Upon entering the meeting room, the meeting participants will want to know what resources are available in the room and how they can be used. Projectors and video conferencing systems are examples of such resources. In addition, the meeting participants may want to access other resources outside of the room, for example the code repository where the code to their program is stored. Finally, it might be useful to access devices, services and resources that other meeting participants bring into the room, such as interactive whiteboards, computers and PDAs.

In order to provide such easy access to services and resources, UbiCollab will have to be able to detect them. This is the job of UbiCollab's Discovery Service. This service supports three different techniques for service discovery (manual, RFID and automatic), and the purpose of this test is to check that this functionality works as planned and that it is sufficient to support such scenarios as above. In addition, the test checks that it is possible to instruct the platform to use a particular Discovery Service.

Manual service discovery is tested by simply starting one of UbiCollab's services. This service will then contact the Discovery Service and is thereby added to the service registry.

Automatic service discovery is tested by adding a UPnP compatible service or resource to the network. This service will then be discovered by the UPnP Discovery plug-in of the Discovery Service and added to the service repository.

For discovering RFID tagged services, on the other hand, an application is needed. This is the purpose of the Service Registry application. Instructing the platform to use a particular tagged Discovery Service is also tested using this application. When doing RFID-discovery, the application reads the service's description URL from the tag and sends it to the Discovery Service. To fulfill the test requirements, the description URL should be successfully transmitted, parsed by the Discovery Service and added to the service repository. To successfully instruct the platform to use a new Discovery Service, the URL should be sent to the Pocket Discovery Service where it should be persistently stored. All subsequent calls to the Pocket Discovery Service should then be forwarded to the new Discovery Service.

6.3.3 UPnP Light control

There is no point in storing handles to resources and services in a repository unless users can retrieve these handles from the repository. Users might also want to search the repository for services and resources matching certain criteria. In the scenario above, the meeting participants might for example want to retrieve all nearby display devices, or search for the meeting room's light control in order to dim the lights for the presentation.

The UPnP Light control application is designed to test this functionality. The application works as a remote control for a certain type of light devices, and does therefore want to know of all such light devices in the system. Consequently the platform should return handles to all such devices when the application requests it. Other services and resources should not be returned as the application cannot use these. This test does therefore check that the platform is capable of performing such a search in its resource repository and return handles to the correct devices. In order to test that the handle returned by the platform actually works, one such light device has been implemented. This application has been named "UbiCollab Light Device" and is capable of turning lights on and off using the X-10 home automation protocol. A Siemens virtual light device is also included in the testbed. For the test requirements to be fulfilled the platform should return handles to these two devices. The application should also be able to use these handles to control the two light devices.

6.3.4 Locator

There are many situations where a user might want to know the position of another user. In the meeting room scenario, for example, some of the participants

may want to know the location of the users that have not showed up for the meeting, where as the users that have not showed up might want to know where the others are so that they can find the way to the meeting room.

Four UbiCollab services come into play in such a scenario: The Discovery Service, The Positioning Service, the Location Service and the Collaboration Service. The Discovery Service is used to find the other services, The Positioning Service is required in order to retrieve the positions, the Location Service is needed to convert these coordinates to more humanly readable locations, whilst the Collaboration Service is necessary in order to retrieve the usernames of the users that are about to be positioned. The latter is important as users do not want to position all the users in the system, but rather those that are somehow related to themselves, i.e., are in the same collaboration instance. In the above scenario, this would mean the other meeting participants.

The purpose of the Locator program is twofold: It tests the functionality of these four services, and it demonstrates how the result from several services can be combined to give the desired result. When the application is started, it uses the Discovery Service to search for Positioning and Location services. Once found, the Collaboration Service is used to retrieve all the users in the same collaboration instances as the current user. The Positioning Service does then return the positions of these users, before the Location Service is invoked with the positions in order to retrieve the corresponding location zones. If successful, the application should display the position and location of the users that are fund in the same collaboration instances as the current user.

One of the most radical architectural changes to the UbiCollab platform in this version has been the decision to create the platform as several small, easily replaceable bundles. This test is therefore especially important as it shows how the result from many such services can be combined to get the same result as from one large service.

The coordinates returned from the Positioning Service should also be correctly displayed on top of the map provided by Google's Google Map service in the Positioning Service Map application. This is vital as it tests the combination of UbiCollab with third-party services.

6.3.5 Slideshow control

Imagine that the meeting participants in the above scenario decide to start the meeting even though not everyone has arrived yet. In order to let the ones that are not in the meeting room follow the presentation, they might want to show the presentation on a display near the other users as well. Both displays should then display the same slides and users in both locations should be allowed to control the slideshow.

This scenario piece is very similar to the scenario that formed the basis for the first version of UbiCollab [25]. The reason for including this as a part of the testbed is to make sure that the new architecture has not come at the expense of supporting such an important scenario. There are also a few other important attributes of this scenario that makes it valuable as part of the testbed. The scenario involves more than one user and thus demonstrates how UbiCollab can be used to facilitate collaboration among several users, even in different locations. In addition, it demonstrates the concept of Collaboration instances and how they can be used to group related items such as users (meeting participants), resources (PowerPoint files) and services (display services).

To test UbiCollab's support for this scenario, a test suite consisting of a Slideshow control application and a UbiCollab Slideshow Service is used. When the user presses the "Get Files"-button in the application, the Collaboration Service returns a description of all the files that are found in the same collaboration instances as the user. This prevents the user from seeing files that are not relevant in the current setting. The user can also use the Discovery Service to search for Slideshow services. The handle to the file can then be sent to these services. The Slideshow Services will retrieve the file from the Data Storage Service and display the slides.

If the test is successful, the display services should display the selected file. If more than one service was selected, both should display the same slides and the user should be able to control both of them with a single click. It should also be possible for users in both locations to control the slideshow.

6.3.6 Data Storage Service Tester

What if one of the users that took part in the meeting wants to review the slides later that day? Or what if one of them has written a meeting résumé that he or she wants to make available to the others?

The questions above were not addressed in the first version of UbiCollab. Users did therefore have to use external tools such as email to distribute information, or they would have to have access to, and know how to publish information on a web server or similar. To solve this problem, the current version includes a Data Storage Service. The service works as a shared storage space where users can store data. The service exposes a web-service interface just like all the other platform components. It is therefore easy for application developers to include functionality for uploading and downloading data to their applications. The Data Storage Service Tester is such an application and is designed specifically to test the functionality of the Data Storage Service. It should be noted that the Data Storage Service is also used by the Slideshow control. However, that application does only test downloading of data. With the Data Storage Service

Tester, on the other hand, both downloading, uploading, deleting and listing of currently stored data can be tested.

To successfully pass the test, data should be uploaded, listed on the service, downloaded and removed without exceptions. The downloaded data should be an exact copy of the data that was previously uploaded.

6.3.7 Login Service

In 1991, Mark Weiser introduced the vision of ubiquitous computing in his article "The Computer of the 21st Century" [26]. In it, Weiser describes a scenario where public computing devices of different sizes are scattered around, much like post-it notes and sheets of paper are today. These devices are, according to Weiser, not intended to have any individualized identity or importance, and could be used by anyone and then just left behind for someone else to pick up. Once picked up, or for the larger ones, approached, these devices temporarily adapt to the current user and display certain personalized information.

The purpose of the Login Service is to simulate such a scenario. To do so, the test suite consists of two components. The Login Service, which is a platform service intended to run on the public display, and the Remote Login application which is supposed to run on an active badge carried around by the user. For the test, however, a laptop is used as a public display whereas a PDA is used as the active badge.

Weiser states that the public device is personalized by either approaching it or by picking it up. In this test we simulate that by scanning an RFID tag attached to the public display. The application will then send its user credentials to the Login Service at the URL read from the tag.

For the test to be successful, the public display should evaluate the username and password combination sent by the Remote Login application. If the combination is valid, the display should show user data such as the username and the user's current collaboration instances. Otherwise, an error message should appear on the public display.

6.4 Practical experiences

The construction of the testbed has also revealed some issues related to development for smaller devices. In the testbed, this type of device was represented by a HP iPAQ h6300 PDA, but it is likely that the experiences gained from the development of applications and platform services for this device will also apply

to other types of constrained devices. This section will therefore briefly present some of these experiences. More on this topic can be found in the Appendix.

6.4.1 Memory handling

The iPAQ used in the testbed has a total memory of 64MB, which is shared between application and storage memory. However, most of this memory is taken up by preinstalled applications that cannot be removed, and necessary drivers. When Java and the Knopflerfish OSGi framework are installed on top of this, the remaining free memory falls dangerously low. As a consequence of this, the PDA is not capable of running two instances of Java at once. This is unfortunate, as it means that Java applications cannot be run on top of the UbiCollab platform (which is also run by Java). If this is attempted, the operating system will automatically shut down the least used Java application in order to save memory. This is usually UbiCollab, as it is run in a background thread. This behavior cannot be overridden, and the solution adapted in this testbed has therefore been to create test applications in C# for the .NET Compact Framework, as these use less memory than Java applications. This works fine in most cases, but for later tests and development a more powerful PDA should be used.

6.4.2 Java

There are currently no implementations of Java for Pocket PC that supports the full Java SE specification. In this testbed, the IBM J9 implementation has been used. This is one of the most popular implementations for Pocket PC. In spite of this, J9 is an incomplete implementation of the Java 1.3.0 specification which was launched on May 8th, 2000, and thus is more than six years old at the time of writing. This causes problems in some situations, as the implementation lacks important functionality. This is especially critical in the cases where a third-party Java library is needed, as these are almost always based on newer versions of Java. In this work, such problems have been solved by either recompiling the particular library for Java 1.3.0, or by adding the missing functionality to the J9 implementation. These fixes have been included on the CD accompanying this report, so that they can be re-used on other devices.

6.4.3 OSGi

Most OSGi frameworks are shipped with, or are capable of downloading, important bundles that enhance and extend the functionality of the framework. Knopflerfish's axis-osgi bundle, a bundle containing the Apache Axis web service server, is an example of such, which enables bundles to expose their interfaces as web service interfaces. Unfortunately, it is not certain that such bundles will run smoothly, even when they are distributed by Knopflerfish. The axis-osgi bundle, for example, includes a version of Apache Axis that is too new to be run by J9. For some reason it is also not compatible with the newest version of

Java (version 5.0). Two new versions of this bundle have therefore been created as part of this work: One version where the internal Axis representation is replaced with a J9 compatible version, and one version that includes the newest Java 1.5 compatible version.

6.4.4 Services on small and constrained devices

Because of the constraints of the smaller devices, the decision of which services to deploy on these devices should be well considered. There is no correct answer to this question, and the answer will vary from device to device, depending on the capabilities of the device and the requirements of the bundle. But the experiences gained from the work on this testbed indicate that low-end current PDA's quickly run out of memory if too many services are deployed on the device. The processing power of these devices is also considerably lower than for larger devices. Processor intensive operations should therefore be executed on larger devices. On the other hand, less processor intensive operations can be executed faster on the local device because data does not have to be sent across the network. Consequently, these services will also be available in the case of a network failure or loss of network signal. Finally, some services have to be run locally. The GPS Positioning plug-in is an example of such, as it has to be located where the GPS receiver is.

Chapter 7

Platform evaluation

The chapter presents an evaluation of the new UbiCollab platform with regards to the requirements found in the autumn report [20]. It also compares the evaluation of the new platform against the evaluation of the old one. Finally, it analyzes the experiences received when testing the platform with the created test applications and services.

In the autumn report, several platforms for collaboration were analyzed; Gaia [13], The ABC Framework [4], Collaborator [21], and the old version of UbiCollab [25, 14, 22]. Summarized descriptions of the requirements can be found in Chapter 2, while full descriptions are found in the autumn report [19].

7.1 Analysis with regards to requirements

7.1.1 Presence management

This requirement is not implemented. Presence management is a very important area of collaboration technology that, especially in the context of this work, is not fully understood. More research is needed here to build a system that would be able to obtain presence information from users while operating in the background and not be obtrusive to the users. Research is also needed on finding a representation of the presence information so that it reflects a user's true presence, for instance including the possibility of having different presence values depending on who is requesting it.

7.1.2 Resource collection

Resource collection is provided through the fully implemented Discovery service. Currently it supports web service services and UPnP devices. While discovery of web service services happens through a service or user initiated registry process, the Discovery service employs a plug-in architecture where automatic discovery

for UPnP devices is provided through a UPnP Discovery plug-in. Automatic removal is also provided for UPnP devices, as long as they disconnect gracefully and send their UPnP Byebye messages. If they do not, and for all other services that do not remove themselves from the Discovery service, the Discovery service tries to contact all discovered services every 30 minutes and purges those it cannot reach. The Discovery service currently supports global discovery by default for web service services, and local discovery for any networks where UPnP Discovery plug-ins are deployed. It does not currently support discovery by location, as there is currently no way of retrieving such information when discovering services automatically, and because of this, specifying such information when registering services has not been implemented. As mentioned, the Discovery service supports services on different technologies, currently web services and UPnP, and other technologies, such as JXTA, can be easily added as discovery modules.

7.1.3 Positioning

A Positioning service fulfilling this requirement is implemented, and is based on the geographic coordinate system with the three values latitude, longitude and altitude. The service employs a plug-in system, where the different plug-in types provide position data from different positioning technologies. Currently only the GPS plug-in, for devices with access to GPS receivers, is fully developed. Plug-ins for GSM and WLAN positioning should be trivial to implement, as long as the developer gets access to such systems for positioning. Since only one type of positioning currently exists, no functionality for switching between positioning technologies, depending on certain criteria such as the user being indoors, has been implemented. It is possible for users to stop the system from being able to position them by merely stopping their GPS plug-ins, and a similar feature where the users are removed from the positioned subjects in a GSM or WLAN positioning mechanism have to be implemented for these. Finally, the service is not developed with the intention of being able to perform any calculations on position data, but leaves this to smaller add-on services or the applications themselves.

7.1.4 Location management

This requirement is fulfilled in the form of the Location service, which is able to perform mappings from positions, in the form of coordinates, to locations, represented by textual names or descriptions. Mapping from locations to positions has been discarded, as this seems to add no value to the system. The original idea was that the platform should be able to answer questions such as "Which users are in the meeting room?", but this feature is better implemented by other mechanisms. Currently, the list of locations with their associated coordinate polygons is very small.

7.1.5 Privacy

The requirement of Privacy has not been implemented, but as its function is added as an aspect in the Identity manager service, some parts of the design are specified. The design is based on letting the user have different virtual identities, and let these be used in different Collaboration instances according to the user's needs for presenting certain information or for complete privacy.

7.1.6 Security

This requirement is not implemented, but its specified mechanisms should be added to any relevant part of the platform; currently the Discovery service, the Data storage service and the Collaboration service. For the Discovery service, the security requirement is twofold: Firstly, usage of services needs to be controlled by an authorization scheme, and secondly, private services should not be visible to unauthorized users in the Discovery service. The first part can be achieved by implementing OASIS' WS-Security[1] specification for web services, the UPnP Security[2] specification for UPnP devices, or similar specifications for other technologies. For the second part, some sort of detailed authorization scheme for the Discovery service entries have to be designed. For the Data storage service and the Collaboration service, accessing files and Collaboration instances should be controlled by an authorization system using credentials such as usernames and passwords.

7.1.7 Persistent data storage

The implemented Data storage service covers the requirement of persistent data storage. Files are currently stored with the format username/appname/filename. Building services, such as versioning control or concurrent document sharing, on top of this service is as easy as having those services implement their needed logic and use the interface of the Data storage service to store and retrieve data. Applications would use the top-level services directly, and would be oblivious to the fact that those services use the Data storage service underneath. Users can also share files directly, by adding them to Collaboration instances which both parties are members of. As mentioned earlier, no security mechanisms for the data storage or retrieval is implemented. The Data storage service is operating system and file system independent, as it only stores and manages the files as a series of bytes. There is currently no functionality such as automatic backup or recovery of destroyed or deleted data, but redundancy can be realized by storing

[1] http://www.oasis-open.org/committees/wss/
[2] http://www.upnp.org/standardizeddcps/security.asp

the same files on more than one Data storage service.

7.1.8 User profile management

User profile management is not implemented, but is designed as a part of the Identity manager service. According to this design, the Identity manager service handles both standard specified user profile entries, such as username and email address, but also domain specific entries, such as a user's favorite movie genre or his or her office hours, that can be useful for certain applications.

7.1.9 Asynchronous communication support

This requirement is not directly implemented, but it is fulfilled by the implementation of the Data storage service, upon which services such as an asynchronous mailbox can be developed.

7.1.10 Scalability

To minimize the possibilities of bottleneck behavior, the platform's core services are capable of running on different hosts. This is made possible by having them exposing their own web service interface, and because they register themselves and are found by other parties through the Discovery service. When it comes to service discovery, it is still not clear how much congestion a great number of UPnP devices will cause, but seeing as the platform components are no longer UPnP devices, as in the old version of UbiCollab, this problem is related to upper level services and applications, and not the platform. The platform still uses a hybrid communication architecture, with the central UbiCollab server holding Collaboration instance data, but with the rest of the platform being based on peer-to-peer communication, resulting in minimal traffic through the central server, unless it also is set up to provide some of the services. Another possible method for decreasing the chances for bottleneck behavior is distributing the network load by running the same services on several hosts. In this case, applications can choose which of the services they want to use after querying the Discovery service, or, by adding some mechanics to it, the Discovery service can return only the services with acceptable load levels.

7.1.11 Platform extendibility

This requirement is mainly fulfilled by OSGi's built-in mechanisms for plug-ins and their interdependency handling, but also because all services expose their own web service interfaces, and developers do therefore not have to add code to any other parts of the system. Application developers have very clear web

service interfaces associated with the platform core services to develop against, although these require further application testing before the APIs can be finalized. Since many of the platform services are designed in a three-tier fashion, where plug-ins define the bottom tier, it is only necessary to develop new plug-ins when aspects from new technology needs to be added. This can for instance be other positioning mechanisms for the Positioning service, new technologies upon which to discover services for the Discovery service, or different ways of storing data for the Data storage service.

7.1.12 End-user programming

This requirement is currently not fulfilled. To fully provide uncomplicated and efficient end-user programming for UbiCollab, an ontology specifying the hierarchy of available services and their inputs and outputs should be created and maintained. Then automatic set-up of services, such as those in an enabled home, could be initiated, based on what other services or functional end-points were available. Applications could then also be created to support all services along complete paths in the hierarchy, only limiting or expanding the functionality depending on the currently available service's depth in the hierarchy. A less automatic method of end-user programming could also be possible. For instance by the means of a graphical user interface representing the currently available services and their functionality as building blocks that can be connected to each other, such as the method used with eGadgets [18].

7.1.13 Device operating system independency

As long as the device is capable of running Java applications, more specifically an OSGi framework, all platform components can be run on it. The most important issue here is that not all versions of Java is capable of running the newest OSGi frameworks, and many untraditional devices can currently only run old versions of Java. When it comes to communicating with the platform, any device that is able to transmit and receive SOAP[3] over HTTP is able to fully use UbiCollab and its services.

7.1.14 Communication channel independency

As mentioned for the previous requirement, any peers wanting to communicate with UbiCollab needs to be able to transmit and receive SOAP over HTTP, something that should be possible even for devices such as mobile phones that may be on very limited networks. Content and service adaptation to devices on limited networks, such as only audio for a video conference, is something that

[3] http://en.wikipedia.org/wiki/SOAP

is left to higher level services (e.g., the video conference service).

7.2 Comparison to evaluation of old UbiCollab

An evaluation of the old version of UbiCollab with regards to the same set of requirements was performed in the autumn report [19]. Following is a set of experiences found after comparing the evaluations.

The old version of UbiCollab had a form of presence management based on how active a user was within their Collaboration instances [25]. This mechanism could work well in settings such as meetings, which the old version of UbiCollab focused on, but not with more general scenarios that the new version is designed to cover. In a Collaboration instance representing an enabled home and its inhabitants, for instance, the inhabitants' degree of activity would not be a clear measure of their presence. Therefore, more research will have to be dedicated on how to retrieve, analyze and represent presence information for the new UbiCollab.

While the old version of UbiCollab was heavily based on UPnP and could only discover UPnP devices, the new version is more general and not based on any technology restricting its discovery possibilities. The new Discovery service supports services exposed as web services inherently, and has plug-in capabilities for supporting other technologies. The capability of automatically discovering UPnP devices is developed and added as a plug-in to the Discovery service, and plug-ins for discovering services on other technologies are easily added in a similar fashion. While the mechanism for manually registering services is mainly intended for web service services, or similar services that cannot be automatically discovered, this feature can be used to manually register services on any type of technology.

As for positioning, the old version had a system that was designed but not fully operational, and position coordinates had to be reported manually by the user to the system. The Positioning service of the new UbiCollab has a similar system, but with a working positioning plug-in for devices with GPS receivers. This plug-in works by communicating with the GPS receiver and passing the position coordinates to the Positioning service whenever the position of that user is requested. Plug-ins for other positioning technologies can easily be added in a similar fashion. The old Positioning service was able to calculate distances between two positions, but this is left to higher level services in the new version of UbiCollab.

The old version of UbiCollab provided user privacy through a Privacy service that made it possible to use the platform anonymously. This is an aspect that is not added to the new version as of yet, but it is designed as a part of the

Identity manager service.

While the old UbiCollab did not have any means for users to persistently store data, the new UbiCollab has a fully functional Persistent data storage service as one of its core platform services.

Both versions of the platform use peer-to-peer communication, along with a dedicated UbiCollab server. As illustrated in Figure 7.1, the difference with regards to scalability is that the old version funneled all traffic to and from the platform through the UbiCollab server, while the UbiCollab server is just another platform service peer in the new version. Since the old platform was based on UPnP, it was also more susceptible to the network congestion UPnP devices can create.

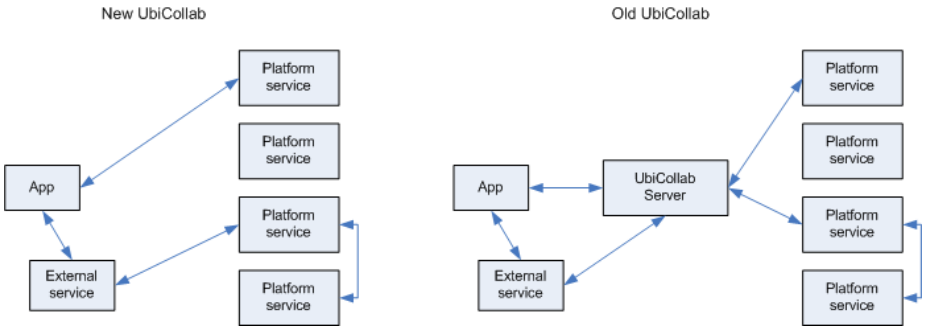

Figure 7.1: Platform communication comparison

One of the main problems of the old version of UbiCollab was that it was hard to extend with new platform services and functionality. This was mainly because, in addition to creating the service and its interface, the developer would need to write the service functionality into the UbiCollab server to make it able to pass traffic to and from the new platform service. While this is a tough problem to start with, it gets harder by the fact that several different developers added such services after the original platform's conception, modifying the same code. In the new version of the platform, the platform service developer only needs to develop the actual service and its interface, and then use OSGi's built-in mechanisms to plug the service into the framework running that part of the platform.

The new platform is built to run within an OSGi framework, which again is based on Java, making the new platform more device operating system independent than the old version, which had to run on devices supporting UPnP. This also goes for external services' and applications' ability to communicate with the platform, as the communication method for the new platform is traditional web service SOAP over HTTP, instead of requiring the devices to support UPnP calls. This fact also makes the new platform more communication channel

independent than the previous version, as SOAP over HTTP is a more general and available form of communication than UPnP calls.

When it comes to the Location service, security, user profile management, and end-user programming, the two versions of the platform are very similar. The only difference with regards to asynchronous communication is that the new version is able to support it easily because of the added Data storage service.

7.3 Evaluation with regards to testbed

Whether or not UbiCollab can be considered a success, is to a large extent determined by the end-users of the system. However, evaluating this aspect cannot be achieved without building applications on top of the platform. This is because UbiCollab is an infrastructure system, and thus is intended to support the construction or operation of other software, and not users directly. This implies that evaluation of the usability of UbiCollab can only be achieved indirectly via applications that are built on top of UbiCollab [7]. This does, however, involve some challenges: How to choose what applications to create to evaluate the platform? And how to make sure the evaluation results can be traced back at the platform and not the application used in the evaluation process?

In order to meet these challenges, the evaluation of UbiCollab has been based on the advices given by Edwards et al. in the article "Stuck in the Middle: The Challenges of User-Centered Design and Evaluation for Infrastructure" [7]. In it, Edwards et al. recommends building several small applications that each test a small part of the core functionality of the platform. The applications described in Chapter 6 are built with this in mind, but in itself this is not enough to guarantee a "correct" and complete evaluation of the platform. Later phases of evaluation will require testing with more heavyweight "real world" applications. Testing will also have to be conducted with actual end-users, both technically experienced and users with limited technical experience, as well as application developers.

Despite of this there are some conclusions that can be drawn from the evaluation of the tests described in Chapter 6.

7.3.1 Programming language independence

Test applications have been built using several different programming languages and runtime environments (e.g., Java, C# /.Net, C#/.Net Compact Framework and PHP). These test shows that UbiCollab is capable of communicating and collaborating with applications regardless of the programming language they are written in. This indicates that the goal of creating UbiCollab as a programming language independent system is close to being fulfilled.

7.3.2 Device and operating system independence

Platform components have been deployed on both a PDA running Windows Mobile 2003 and on two different laptop computers running Windows XP. Even though more testing is needed, this indicates that the platform can operate independently of operating system and devices. On the other hand, the tests do also show that some platform services are too resource-draining to be effectively run on smaller devices. Care should therefore be taken when deploying platform components so that the components are deployed on suitable devices.

7.3.3 Collaboration instances

One of the fundamental concepts in the UbiCollab platform is the use of collaboration instances to group related items such as users, resources and services. Some of the tests (i.e., the Locator and Slideshow control) have proved that this concept works as intended, at least in a small scale.

7.3.4 Collaboration support

The use of UbiCollab and the concept of Collaboration instances to support collaboration between users have also been successfully tested. In the Slideshow control test, users could share Power Point files with each other and also control the same slideshow from two different locations. This is an encouraging result, but again more testing is needed in order to make sure this does also work as intended in a real world setting.

7.3.5 Usage of OSGi

The tests have also showed that OSGi fulfills its promises. Bundles can indeed be started, stopped, modified and redeployed on another device without having to restart the framework. On the other hand, these tests have only been conducted with the Knopflerfish OSGi framework. It is therefore not certain whether all services work properly in other OSGi frameworks.

7.3.6 Three-tier architecture

Quite a few of the new UbiCollab services have been based on a three-tier architecture with plug-ins as the lower tier. Testing this has had a high priority as this architecture is new to this version of UbiCollab. The results from the initial tests were very promising. They showed that new plug-ins could be built and deployed in the platform without having to make any changes to the code of the parent component. The tests also showed that one plug-in can take over for another one if that one goes down. This is completely transparent to the applications, which are not affected by this. On the other hand, the tests did also confirm that OSGi cannot detect changes in a plug-in's status (i.e., started, stopped, modified) if that plug-in is found in another OSGi environment. This

means that some services will have to base its plug-in functionality on other technologies, such as web services.

7.3.7 Combining the effort of several services

Some tasks only require the services of one single platform component, but in many cases the tasks are too complex to be handled by a single service. It is therefore crucial to make sure the result from more than one service can be combined into a more powerful result. Three such tests were conducted, and all succeeded. The locator application tested the situation where the application uses the result from one service as input to another, the Positioning Service and its plug-ins tested the situation where a request is forwarded from one platform component to another, whereas the Positioning Service Map tested the combination of a UbiCollab service with a third-party service. The success of these tests indicates that this important feature of the platform works as intended

7.3.8 Resource collection

Resource collection is another important aspect of UbiCollab. The tests have proved that UbiCollab is capable of handling resource collection both by manually adding them to the repository, by collecting them via RFID and by automatic detection. The latter does, however, only work with UPnP devices. Future research should investigate means of automatically collecting web-service based services. In addition, more plug-ins should be built for the Discovery Service so that for example JXTA [Footnote] is supported. The search functions used when querying the service repository for certain types of services have also proved sufficient in the test cases. However, this is probably not sufficient in a "real-world" deployment and the use of some sort of ontology for resources and services should be considered.

Chapter 8

Conclusions

8.1 Contributions

Due to the contributions from several projects and theses over the last few years, UbiCollab has achieved many of its goals, such as automatic discovery of services and the collaborative benefits of the abstract Collaboration instance model. However, the study conducted in the autumn of 2005 [19] showed that UbiCollab would not be able to reach its vision of being a flexible and extendible platform for ubiquitous collaboration with its existing architecture. This thesis has therefore focused on completely re-designing the architecture of the UbiCollab platform so that it will come closer to fulfilling its vision. The design process has been based on the conclusions from the autumn report [19], experiences from the old version of UbiCollab and the use of new technologies such as OSGi. The process has also involved a thorough evaluation of the services UbiCollab should provide, and a restructuring of the platform's application programming interface (API). Many of the identified services have been fully implemented, whereas other services have been partially implemented with instructions on how the service implementation can be completed. Finally, a testbed consisting of various resources and numerous small test applications has been constructed. This testbed has been used to evaluate the core functionalities of the platform in addition to being a valuable tool for demonstrating the use of UbiCollab.

More specifically, the contributions are as follows:

A new platform architecture has been designed. This new design has made extensive use of OSGi for dependency handling and plug-in management, which has resulted in a platform that is more flexible and easier to extend with new services. The new platform services have also been carefully selected in order to correct the shortcomings identified in the autumn report. A detailed description of the new architecture and a list of the new platform services can be found in Chapter 3 and Chapter 4, respectively.

A set of platform service APIs have been given in Chapter 5. These have been constructed based on the platform requirements found in the autumn report [19], which were compiled through a process of analyzing collaboration in many different scenarios, and comparing a set of collaboration platforms. The new platform API should therefore be better suited to support ubiquitous collaboration in a variety of environments and settings.

A working prototype consisting of the most essential platform services has been built. The prototype constitutes a solid basis for further research and development, and is also valuable for evaluating the feasibility of the new platform architecture, testing new platform services and demonstrating the use of UbiCollab. Details of the prototype implementation are given in Chapter 5.

The platform has been evaluated using two different techniques. The first evaluated the platform with regards to the requirements from the autumn report [19], whereas the second one involved the creation of a testbed with several small applications and external services that each tested a piece of the core functionality of the platform. As the old version of UbiCollab was evaluated with regards to the same requirements in the autumn report, a comparison of the evaluations has also been performed. The testbed used in the evaluation is described in Chapter 6, whereas the results of the evaluation as well as the comparison with the old evaluation results are found in Chapter 7.

Finally, this work has reported on experiences with the testbed hardware and software, and the fixes that had to be made to it in order to deploy and run the platform and the test applications on different devices in the testbed. These experiences and fixes are presented at the end of Chapter 6, and further explained in the Appendix.

8.2 Evaluation

The goal of this thesis has been to re-design the platform architecture of UbiCollab, so that it will be more flexible and easier to extend with new services and technologies. Initial testing, and comparison of the new design with the old one, strongly indicates that this has been accomplished. However, the platform has not yet been tested with actual end-users or application developers, nor has it been tested over a long period of time in a "real world" setting. According to Edwards et al. [7], this is not much of a problem, though, as this kind of testing should only be conducted in the later stages of development.

The usage of OSGi plays an important role in the new platform and is crucial in achieving the goal of flexibility and extendibility. On the other hand, this does also make the platform dependant on OSGi and forces all devices running

platform components to install it. This could potentially become a problem for the smallest and most constrained devices, as they might not have sufficient resources to run the framework. As the platform is dependant on OSGi, it would also be difficult to port it to another technology in case the usage of OSGi is no longer wanted. However, this problem would occur no matter what technology the platform was based on, and can therefore not be considered a serious disadvantage.

The platform API has been built from the requirements identified in the autumn report [19], and refined by stepping through the scenario descriptions in the same report, looking for events that should be translated into service functionality. Finally, the implemented part of the API has been evaluated with test applications in the testbed. This approach has been chosen in order to make sure the API is as correct and complete as possible. Nevertheless, there might still be situations and scenarios that are not fully covered by functionality in the current API. This is due to potential errors in the requirements forming the basis for the API, and uncertainty as to whether the scenarios from which these requirements were extracted constitute a sufficient approximation to the domains UbiCollab is supposed to cover. The API for the non-implemented services are particularly vulnerable as these have not been evaluated in the testbed. This is, however, not such a serious problem, as the new architecture makes it a lot easier to extend the platform with new functionality at later stages. Platform components can also easily be removed from the platform, modified and re-inserted with minimal disturbance to the system.

The current prototype has been designed to run on several different devices. Unfortunately, developing for some devices has proved to be far more cumbersome than expected. This is mainly due to having to run old versions of Java and the Knopflerfish OSGi framework, but also due to limited resources such as memory and processing power. In the current implementation these challenges have been solved, but the process of doing so required complex fixes to many of the native Java, OSGi and third-party libraries. This leads to concerns as to whether some developers will find it too hard to develop new services for the platform. If this is the case, it will conflict with the requirement of extendibility. More research is therefore needed to investigate limitations of the most common families of devices and to find the best suited implementation of these software libraries for each group of device. In the long-term, however, it is likely that this problem will diminish significantly as new Java libraries tailored for small devices will become more common.

Edwards et al. [7] raises the question of how a designer can evaluate features of an infrastructure system, such as UbiCollab, without knowing about its applications and users. This is a difficult problem to solve and there are few reported experiences in the human computer interaction literature on this topic [stuck in the middle]. This thesis has tried to overcome this problem by evaluating the platform using two different techniques. In this way, problems missed using

one technique might be identified using the other one and thus keep errors at a minimum.

In order to make the platform services run on the smaller devices, some modifications had to be made to supporting code libraries such as for instance the IBM J9 Java implementation and Knopflerfish. Special care has been taken to make sure these modifications do not affect the rest of the libraries' code.

8.3 Further work

The dependency handling and plug-in management features of OSGi have proved to be extremely valuable for UbiCollab. Unfortunately, these features, such as the service tracker mechanism, do not work across different OSGi frameworks. This is a serious disadvantage which is further aggravated by the highly distributed architecture of the platform. The current solution to the problem is to use web service calls to communicate with services deployed in other OSGi frameworks. However, web services lack these important plug-in and dependency management features. Further research should therefore try to find ways of communicating with OSGi calls across different OSGi frameworks, or possibly find alternative protocols that provide the same features or is able to simulate the same mechanisms.

The current prototype does not include all the intended services and planned functionality. Some services are only partly implemented whereas others are not implemented at all. Consequently, the current prototype does not meet all the requirements from the autumn report. These services will therefore have to be implemented and evaluated before the platform is finalized. Additionally, some of the unfulfilled requirements can not be met by simply adding another service. The security requirement is an example of this, where security mechanisms will need to be added to components such as the Discovery service, the Data storage service and the Collaboration service.

Further evaluation of the framework should also be conducted. This thesis has evaluated the platform against the requirements from the autumn report, compared the result with the evaluation result of the old version and tested the platform with several service consuming applications. In spite of this, more test-and-refine cycles are required to improve the functionality and APIs of the components, particularly involving application developers creating application that consume services the platform provides.

This thesis has expressed concerns regarding the difficulty in developing platform services for small, constrained devices. Further research should therefore look into ways of making this job easier. This could for instance include detailed guidelines, best practices, Java libraries tailored for particular device families

and development toolkits.

Bibliography

[1] J. E. Bardram, "Supporting Mobility and Collaboration in Ubiquitous Computing," Center for Pervasive Computing, Aarhus, Denmark, CfPC 2003-PB-38, 2003.

[2] Federico Bergenti, Socrates Costicoglou & Agostino Poggi, "A Portal for Ubiquitous Collaboration," in *Conference on Advanced Information Systems Engineering*, 2003.

[3] B. Cameron et al., "Where does groupware fit?" *The Forrester Report: Software Strategies*, Vol. 6, No. 3, June 1995.

[4] H. B. Christensen & J. E. Bardram, "Supporting Human Activities - Exploring Activity-Centered Computing," in *Proceedings of Ubiquitous Computing*, 2002, pp. 107-116.

[5] N. Davis & H. Gellersen, "Beyond Prototypes: Challenges in Deploying Ubiquitous Systems", *IEEE Pervasive Computing*, Vol. 1, pp.26-35, 2002.

[6] Monica Divitini, Babak A. Farshchian & Haldor Samset, "UbiCollab: Collaboration support for mobile users," in *Proceedings of the 2004 ACM Symposium on Applied Computing*, 2004, pp. 1191-1195.

[7] W. Keith Edwards et al., "Stuck in the Middle: The Challenges of User-Centered Design and Evaluation for Infrastructure," in *Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI'03)*, 2003, pp. 297-304.

[8] C. A. Ellis, S. J. Gibbs & C. L. Rein, "Groupware: Some Issues and Experiences," *Communications of the ACM*, Vol. 34, pp. 38-58, January 1991.

[9] Kate Erlich, "Designing Groupware Applications: A Work-Centered Design Approach," in *Computer Supported Collaborative Work*, Vol. 7, *Trends in Software*, M. Beaudouin-Lafon, Ed., John Wiley & Sons, 1999, pp. 1-28.

[10] G. H. Forman & J. Zahorjan, "The Challenges of Mobile Computing," *IEEE Computer*, Vol. 27, pp.38-47, April 1994.

[11] Pedro Goncalves, "UbiClient: A mobile client for an ubiquitous collaborative environment," M.S. Thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2004.

[12] Mike Hazas, James Scott & John Krumm, "Location-Aware Computing Comes of Age," *IEEE Computer*, vol. 37, pp. 95-97, February 2004.

[13] Department of Computer Science, University of Illinois at Urbana-Champaign, "Gaia: Active Spaces for Ubiquitous Computing," http://gaia.cs.uiuc.edu/index.html.

[14] Børge Setså Jensen, "Location-aware service for the UbiCollab platform," M.S. Thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2005.

[15] R. Johansen, *Groupware: Computer Support for Business Teams*, New York: The Free Press, 1988.

[16] Paul Luff & Christian Heath, "Mobility in Collaboration," in *Proceedings of the 1998 ACM conference on Computer Supported Cooperative Work*, 1998, pp. 305-314.

[17] Kalle Lyytinen & Youngjin Yoo, "Issues and Challenges in Ubiquitous Computing," *Communications of the ACM*, Vol. 45, pp. 63-65, December 2002.

[18] Panos Markopoulos, Irene Mavrommati & Achilles Kameas, "End-User Configuration of Ambient Intelligence Environments: Feasibility from a User Perspective," in *Second European Symposium on Ambient Intelligence*, 2004, pp. 243-254.

[19] Christian H. Mosveen & Andreas Brustad, "UbiCollab: Evaluation and requirements re-engineering," Depth study, Norwegian University of Science and Technology, Trondheim, Norway, 2005.

[20] The OSGi Alliance, *OSGi Service Platform Core Specification*, Release 4, 2005.

[21] Agostino Poggi et al., "Collaborator - A collaborative system for heterogenous networks and devices," presented at International Conference on Enterprise Information Systems, Angers, France, 2003.

[22] Hans Steien Rasmussen & Anders Magnus Braathen, "Preserving privacy in UbiCollab: Extending privacy support in a ubiquitous collaborative environment," M.S. Thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2005.

[23] Jörg Roth, "Seven Challenges for Developers of Mobile Groupware," in Workshop *Mobile Ad Hoc Collaboration*, CHI 2002, Minneapolis, 2002.

[24] D. M. Russel, N. A. Streitz & T. Winograd, "Building Disappearing Computers", *Communications of the ACM*, Vol. 48, pp.42-48, March 2005.

[25] Christian Schwarz, "UbiCollab - Platform for supporting collaboration in a ubiquitous computing environment," M.S. Thesis, Norwegian University of Science and Technology, Trondheim, Norway, 2004.

[26] Mark Weiser, "The Computer for the 21st Century," *Scientific American*, pp. 94-100, 1991.

Appendix A

Software fixes

A.1 Java Communications API for Pocket PC

Sun does not provide a Pocket PC compatible version of its Java Communications API[1], however several third-party implementations exist. Many of these implementations were tested with the HP iPAQ h6300 PDA, but none appeared to work. The reason for this was eventually traced to the CommPortIdentifier class. It appears that all these implementations are based on Sun's original implementation. This implementation is quite old though, and the CommPortIdentifier class does therefore use a String constructor that has since become deprecated. This is normally not a problem, as it is perfectly legal to use deprecated methods. However, IBM has decided to not implement any deprecated methods in their J9 Java implementation. Consequently, the Communications API does not work in conjunction with J9. This problem was corrected by decompiling the Communications API and replacing the deprecated constructor with a non-deprecated one. The API was then recompiled for Java 1.3.0. The fixed version of the Communications API can be found on the CD accompanying this report.

A.2 axis-osgi bundle for J9

The axis-osgi bundle is necessary in order for services to be able to expose their interfaces as web service interfaces, and is essentially the Apache Axis web service server, wrapped as an OSGi bundle. The problem with this bundle was that its version of Apache Axis used Java 1.4-specific functionality, and consequently did not work with the J9 implementation which is based on Java 1.3.0. This problem was solved by replacing the internal Apache Axis implementation with an older, 1.3 compatible version. In addition, the namespace `org.apache.axis.encoding.ser` had to be included in the bundle's export list

[1] http://java.sun.com/products/javacomm/

in order for it to work properly, whereas the `java.xml.soap` namespace had to be removed from the import list, as this was erroneously added to both the import and export list.

A.3 axis-osgi bundle for Java 5.0

For some reason the axis-osgi bundle did not work with Java 5.0 either. It seems this was due to backwards compatibility issues with Java 5.0 and Java 1.4, where axis-osgi's version of Axis did not work with Java 5.0. In the same way as for the J9 version, this was solved by replacing the internal Axis implementation with another version. This time it was replaced with the newest Axis implementation available. The same changes in the bundle's manifest file as for the J9 version had to be made for this version as well.

A.4 WSDL parsing

The dynamic proxy generation process in the Positioning Service requires the WSDL file of the plug-in to be parsed. The wsdl4j library[2] is used for this purpose. This works fine with Sun's Java implementation (version 1.4 or higher), but not for the J9 implementation. This is because J9 does not include a content handler for the content type: `text/xml`, and is therefore not able to determine the type of content found at the URL of the WSDL file. Adding a content handler for `text/xml` to the J9 implementation turned out to be extremely difficult. However, a thorough examination of the working Java implementation revealed that the code simply opens an input stream to the URL as soon as it has determined that the content is XML. As it is known that WSDL files are always described in XML, this problem can be solved by simply bypassing the content handling process and instead opening the input stream to the URL directly. This was accomplished by modifying the `StringUtils` class of wsdl4j as shown in Listing A4.1.

```
com.ibm.wsdl.util.StringUtils

Public static InputStream getContentAsInputStream(Url url) {
   ...
   //Object content = url.getContent(); // Old version
   Object content = url.openStream(); // UbiCollab fix
   ...
   return (InputStream)content;
}
```

Listing A4.1

[2] http://sourceforge.net/projects/wsdl4j

To make sure this fix does not introduce errors to other parts of the system, (i.e., components trying to parse other types of content) the modified wsdl4j library has been included as part of the Positioning Service. This prevents other services from unintentionally using the modified wsdl4j library.

A.5 Dynamic proxy generation

Because of the dynamicity in the environment in which UbiCollab is found, the Positioning Service is forced to create web service proxy classes to its plug-ins at runtime. This process involves parsing of the WSDL files to see if they contain the correct method, the actual proxy generation, and finally the casting of the proxy to a `PositioningPluginService` interface. Once the proxy is casted to an object implementing this interface, methods can be called on the plug-in as if it was a local Java object.

The problem arises when the proxy is to be casted to a `PositioningPluginService` object. This casting is the last part of the proxy generation process which is done by the `org.apache.axis.client.Service` class. As this class is part of the axis-osgi bundle, it is loaded into the Java Virtual Machine by the same ClassLoader that was used when this bundle was loaded and started. This is a different ClassLoader from the one used to load the classes in the Positioning Service. Consequently, the ClassLoader used in the Service class does not know of the `PositioningPluginService` interface and thus cannot cast the proxy to such an object.

This problem was solved by overloading the method that generates the proxy so that it takes a ClassLoader object as an extra input parameter. When the method is called, the Positioning Service's ClassLoader is passed as a parameter and used to load the `PositioningPluginService` interface. Before the method returns, the original ClassLoader is restored. This modification to the axis-osgi bundle should not affect other components in the system as no changes have been made to any of the original methods. The only difference is the addition of another method that takes a ClassLoader as an extra input parameter.

A.6 Missing functionality added to J9

As J9 is an implementation of the Java 1.3.0 specification, a large amount of important functionality is missing, especially related to XML parsing. In order to compensate for this, a special library with important functionality has been created and tailored for the PDA. This library contains the namespaces: `org.xml.sax`, `org.w3c.dom`, `javax.xml.parsers` and `javax.naming`. It has been created by recompiling the source files from newer Java versions to Java 1.3 compatible classes. This new library has been named *PocketParser.jar* and can be found on the CD accompanying this report.

Appendix B

Installation guide

B.1 PDA

B.1.1 Install instructions

This section gives instructions on how to install UbiCollab and its test applications on a Pocket PC compatible device.

The following components have to be installed before UbiCollab and the test applications can run:

Microsoft .Net CF 2.0

The Microsoft .NET CF 2.0 can be found in the following locations:

URL http://www.microsoft.com/downloads/details.aspx?FamilyID=9655156b-356b-4a2c-857c-e62f50ae9a55&DisplayLang=en

Install the file on a standard computer. The installation process will automatically copy the framework to the PDA if the PDA is connected to the computer and Microsoft Active Sync (version 4.1 or later)[1] is running.

IBM J9

IBM J9 can be found in the following location:

URL http://www-306.ibm.com/software/wireless/weme/

IBM J9 is integrated in the WebSphere Everyplace Micro Environment. This is a commercial software suite, but a free trial version is available. Download this

[1] http://www.microsoft.com/windowsmobile/downloads/activesync41.mspx

version and follow the instructions. Make sure to select the Connected Device Configuration (CDC) with the Personal Profile (PPRO) when instructed.

Knopflerfish Tiny OSGi framework

The Knopflerfish Tiny OSGi framework can be found in the following locations:

URL http://knopflerfish.org/download.html

CD folder Knopflerfish Tiny

It is recommended to use the CD version, as this is complete with the UbiCollab services and correct property files.

The Knopflerfish framework can be copied directly to the PDA. Put the three folders in the "Knopflerfish Tiny" folder in the root directory of the PDA.

Java Communications API

An implementation of the Java Communications API for Pocket PC can be found in the locations below. Make sure to download the Communications API and not the JVM if the web URL is used.

URL http://www2s.biglobe.ne.jp/~dat/java/project/jvm/index_en.html

CD folder Java Comm API for Pocket PC

To install put:
javax.comm.jar in J9-Root/PPRO10/lib/jclPPro10/ext
javax.comm.properties in J9-Root/J9/PPRO10/lib
javaxcomm.dll in /Windows

IDBlue Bluetooth RFID Pen drivers

The drivers for the IDBlue RFID Pen can be found in the following locations:

URL http://www.cathexis.com/secure/idblue.aspx (registration needed)

CD folder IDBlue Software

Install the IDBlue Software Suite on a standard computer. Baracoda Manager for ARM PocketPC (Widcomm 1.4) and Franson Bluetools for ARM,X86 PocketPC (Widcomm and Microsoft) have to be installed on the PDA. They can be found under "Cathexis Innovations" ▷ "Bluetooth Support Installers" ▷ "PPC" on the Windows Start menu, after the installation is complete. These two libraries are also referenced in the code of the applications that utilize the RFID pen. If the references in these projects no longer point to the correct location, update them to point to these two DLLs.

UbiCollab configuration files

The UbiCollab configuration files can be found here:

CD folder UbiCollab Services configuration files

Copy the configuration files to the root directory of the PDA.

B.1.2 How to start UbiCollab and the test applications

After the installation is complete, UbiCollab can be started by clicking the run.lnk file in the /knopflerfish directory, or by running the command in the run.txt file.

The test applications can be started by copying them from the CD to the PDA and launching the corresponding EXE-file. Another approach is to open the source code in Visual Studio and select "deploy to device".

B.1.3 How to install new services for UbiCollab

Put the JAR-file of the new service in the /iPAQ File Store/bundlefiles folder. Add "-install" and "-start" commands to the remote-init.xargs file in the /iPAQ File Store folder.

B.2 PC

B.2.1 Install instructions

When installing UbiCollab on a PC, the prerequisites depend on which bundles are to be installed. This installation guide assumes that all bundles will be installed, and thus covers all prerequisites.

Java 5.0 with MySQL driver

Java (version 5.0 or newer) is required in order to run UbiCollab, and can be downloaded from the following location:

URL http://java.sun.com/j2se/corejava/index.jsp

In addition, a MySQL driver for Java is needed. This can be downloaded from the following location:

URL http://www.mysql.com/products/connector/j/

MySQL

If an external database server is not available, MySQL will have to be installed. The latest version can be found here:

URL http://dev.mysql.com/downloads/

The database server needs to have a database called ubicollab, with username ubi and no password.

The database needs to keep these tables:

discovery		
id	INT	Sequential table-specific ID number
tagId	VARCHAR(40)	ID string on RFID tag
uuId	VARCHAR(100)	ID string for UPnP devices
name	VARCHAR(100)	Name of the service
type	VARCHAR(100)	Type of the service (typically a namespace)
protocol	VARCHAR(10)	The service protocol (ws, upnp, etc.)
descriptionUrl	VARCHAR(200)	URL to the description XML
serviceUrl	VARCHAR(200)	URL to the service XML
owner	VARCHAR(100)	Owner of the service
discovered	TIMESTAMP	Time of discovery

persons		
id	INT	Sequential table-specific ID number
username	VARCHAR(20)	Person's username
fullName	VARCHAR(100)	Person's full name
added	TIMESTAMP	Time of addition

collabinsts		
id	INT	Sequential table-specific ID number
collabInstId	VARCHAR(40)	ID string of collaboration instance
name	VARCHAR(100)	Name of the collaboration instance
creator	VARCHAR(20)	Creator of the collaboration instance
created	TIMESTAMP	Time of creation

personcollabinst		
id	INT	Sequential table-specific ID number
username	VARCHAR(20)	Person's username
collabInstId	VARCHAR(40)	ID string of the collaboration instance
coupled	TIMESTAMP	Time of coupling

filecollabinst		
id	INT	Sequential table-specific ID number
fileId	VARCHAR(200)	ID string of the file
collabInstId	VARCHAR(40)	ID string of the collaboration instance
coupled	TIMESTAMP	Time of coupling

servicecollabinst		
id	INT	Sequential table-specific ID number
descriptionUrl	VARCHAR(200)	URL to the description XML
collabInstId	VARCHAR(40)	ID string of the collaboration instance
coupled	TIMESTAMP	Time of coupling

Knopflerfish OSGi framework

UbiCollab is based on the Knopflerfish OSGi framework and thus requires this framework to run. In addition, the Knopflerfish optional bundles package is required. Both can be downloaded from the following location.

URL http://knopflerfish.org/download.html

From the optional bundles package, the Commons-Logging, and the axis-osgi bundles are required. However, there is a bug in the axis-osgi bundle that makes it incompatible with Java 5.0. Until a new version is available, a fixed version of the bundle can be used. This bundle can be found here:

CD folder Modified axis-osgi bundle/For Java 5.0

UbiCollab configuration files

The UbiCollab configuration files can be found here:

CD folder UbiCollab Services configuration files

Copy the configuration files to this folder: knopflerfish-root/knopflerfish.org/osgi

B.2.2 How to start UbiCollab

UbiCollab can be started on a PC by starting the Knopflerfish OSGi framework. This is done by launching the framework.jar file.

B.2.3 How to install new services for UbiCollab

New services can be added to UbiCollab by starting them in the Knopflerfish OSGi framework. The UbiCollab services can be found in the following location:

CD folder UbiCollab Services bundles