A service discovery and automatic deployment component-based software infrastructure for Ubiquitous Computing

Areski Flissi, Christophe Gransart, Philippe Merle

A Service Discovery and Automatic Deployment Component-Based Software Infrastructure for Ubiquitous Computing[1]

Areski Flissi[1], Christophe Gransart[2], Philippe Merle[3]

[1] LIFL / CNRS,
Université des Sciences et Technologies de Lille (USTL)
59655 Villeneuve d'Ascq, France
Areski.Flissi@lifl.fr
[2] INRETS-LEOST,
20 rue Elisee Reclus, BP 317
59666 Villeneuve d'Ascq, France
Christophe.Gransart@inrets.fr
[3]LIFL / INRIA Futurs,
Université des Sciences et Technologies de Lille (USTL)
59655 Villeneuve d'Ascq, France
Philippe.Merle@inria.fr

Abstract. Software applications running on mobile devices are more and more needed. These applications have strong requirements to address: device heterogeneity, limited resources, networked communications, and security. Moreover it is required to have appropriate application design, discovery, deployment, and execution paradigms. These requirements are similar to those of any ubiquitous computing application. In this paper, we present a component-based software infrastructure to design, discover, deploy, and execute ubiquitous computing contextual applications. Applications are designed as assemblies of distributed software components. These assemblies are dynamically discovered according to end-users' physical location and device capabilities. Then appropriate assemblies are automatically deployed on users' devices. Ubiquitous contextual services and our software infrastructure are built on top of the OMG's CORBA Component Model (CCM) and are implemented using the OpenCCM platform. As illustration, a service to get information about departure trains is described in detail.

1 Introduction

Software applications running on mobile devices are more and more required. This new challenge is now called ubiquitous computing. Lot of problems must be taken into account during development of these ubiquitous computing applications.

[1] This work is completed with the support of Nord/Pas-de-Calais Regional Council and the Regional Management of Research and Technology of Nord/Pas-de-Calais within the framework of the "Contrat Plan Etat/Région".

Terminals used are generally heterogeneous: users can achieve their tasks using a laptop with wireless network card or using a PDA or a smart phone. These terminals have different operating systems and limited hardware resources like RAM or CPU. Moreover mobility is difficult to take into account in applications. Very often, when user goes out of the wireless network coverage, this event generates an error in the application. Last but not least, security in wireless distributed applications is also a big challenge.

Currently, there is no global solution to build ubiquitous contextual applications. Indeed, existing solutions only address the problems of service discovery and deployment. Service design and implementation must be done separately with another technology than that used to build the discovery and deployment functionalities. In this paper, we present a global solution through a component-based software infrastructure to easily design, discover, deploy and execute applications on mobile devices. However, we only focus on applications running in contextual environments like a railway station. Security, disconnection and lost of connectivity are not considered in this paper.

In our approach, a service is deployed on a distributed architecture and running on several equipments (mobile and fixed). Services are designed as assemblies of distributed software components. These assemblies are dynamically and contextually discovered according to end-users' physical location and device capabilities thanks to a service registry. Then appropriate assemblies are automatically deployed to users' devices using a service activator. At runtime, interactions between the components are done using distributed middleware. Ubiquitous contextual services and our software infrastructure are designed on top of the OMG's CORBA Component Model (CCM) [1]. A prototype is implemented using the OpenCCM [2] platform, an open source Java-based CCM implementation.

The reminder of this paper is organized as follows. Section 2 describes an example which will be realized with our platform and the challenges to address in the context of ubiquitous computing. Section 3 presents the design principles to develop a ubiquitous contextual service with distributed software components. These principles are illustrated on the departure train example. Section 4 details the ubiquitous computing software infrastructure composed of discovery and deployment functionalities, especially service registry and activator components. Section 5 describes the implementation of our software infrastructure and discusses about the current limitations. Section 6 presents some related works. Finally, Section 7 concludes this paper and gives some future work perspectives.

2. The context

This section presents an example of applications that we want to realize. Next, we present several problems related with ubiquitous contextual applications.

2.1. An example of contextual applications

To illustrate ubiquitous contextual services, let us take the following example. A commuter with a wireless PDA is arriving into a railway station. When he/she is inside of the hall, his/her PDA automatically knows where his/her owner is and launches services to obtain some information about the departure platform, train schedule, etc. This service can be used with several user interfaces (see Figure 1): with a GUI or with a text to speech interface.

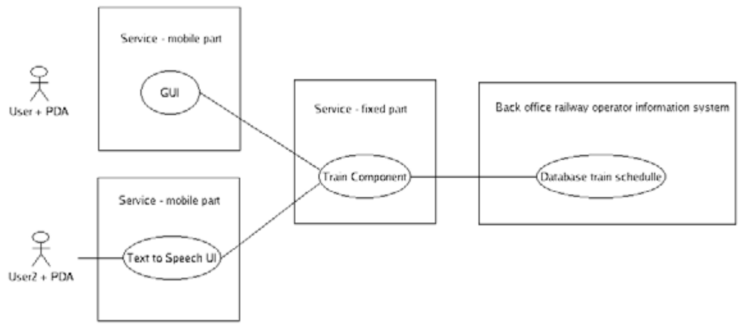

Fig. 1. The Train Service Use Case.

2.2. The challenges

To realize this simple scenario, we must take into account several issues related to ubiquitous and contextual applications:

- **Heterogeneity of devices**
  Devices which could be used are various: user can access to services from a laptop, a PDA or a smart phone. These devices run different operating systems (Windows, Linux, WinCE, PalmOS, Symbian, etc.). Concerning the wireless link, several technologies are also available as Wi-Fi (IEEE 802.11a/b/g) or Bluetooth for short/medium range network, GPRS or UMTS for national coverage.
- **Low Resources**
  Power consumption is a major problem with mobile devices. Moreover, PDA and smart phones have less memory and smaller processor than laptops. Network connectivity is not always available. These points will modify the way to design applications.
- **Distributed Applications**
  The service to get information about the departure platform is distributed on several computers. The graphical user interface is running on the PDA and the process to extract data from the train operator information system is running on fixed computers. Communication between fixed part and mobile part is achieved through the wireless network. Service implementation can be achieved using a communication middleware (CORBA, messaging, etc.).
- **Security**

Security is also a major issue in this kind of applications. All the actors (mobile terminals, infrastructure, and servers) are concerned. To achieve security, several solutions based on authentication and cryptography need to be integrated.

- **Service Discovery**

  The next problem is how a user knows that some services are available in a particular location. Moreover, code of available services to deploy will be different if the computer is a black and white PDA or a color laptop for instance. To make this choice, the system needs some information about the client terminal. According to this information, it can present a subset of services which can potentially run on the user's terminal.

  The discovery process can be done by the user: he/she explicitly sends a request to the environment to check if some services are available. The opposite solution is that the environment sends periodically information about available services. Then the user is automatically notified that some services exist.

- **Service Deployment**

  Once the user knows that an interesting service is available then the infrastructure must deploy the code on the user's device. Deployment consists to fetch software components from a repository, to download them, to instantiate, configure, and interconnect components. In our context, applications must be incrementally deployed: components running on fixed computers are deployed in a preliminary step, and then components running on user's terminal are deployed on user's demand.

  When a user comes back to a location where he/she used previously a service, does he/she have to redo the discovery and deployment process? It would be interesting that its terminal maintains information about services previously used and create a kind of service information profile.

3 The Model for Ubiquitous Contextual Services

This section presents our model for designing ubiquitous contextual services. Firstly, the principles of the underlying software component model used in our approach are discussed. Then the component-based design of the train service is presented.

3.1 The distributed software component model

In our model, a ubiquitous contextual service is a composition of distributed and interconnected software components. The underlying component model is based on the definition given by Clemens Szyperski in [3]: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties". In this definition, an interface means a set of method signatures. A signature captures a method name, a set of parameter types, a set of exceptions, and a result type. Context dependencies encompass all metadata describing the required context to run the component, e.g. binary libraries and other components required by the implementation of the component. However we enhance this definition with the notions of distribution and interconnection.

A ubiquitous contextual service is deployed and run on a distributed system composed of several fixed and mobile computers. Then its components are distributed over the system: Each software component is assigned on a node in the system. In order to support component interactions, data exchanges and component behavior synchronization, two kinds of distributed interaction patterns are needed at least[2]: 1-to-1 synchronous invocations and 1-to-many asynchronous notifications. With the first interaction pattern, a client component calls methods provided by a server component in a synchronous way, i.e. the client waits until the server replies. For instance, this allows a client component to obtain a data stored or computed by a server component (e.g. get the list of departure trains) or to request the execution of some behavior of a server component (e.g. add a new arrival train). With the second interaction pattern, a producer component sends data to consumer components in an asynchronous way, i.e. the producer and consumers work in parallel. For instance, this allows a producer to broadcast the new value of a data to all interested consumers or to notify a new event like an alarm (e.g. notify the departure or arrival of a train).

In order to support these two interaction patterns, the components need to be interconnected. For this purpose, each component provides interaction ports. Four kinds of ports are available[3]: *facet* identifying an interface provided by a server component and its methods are invoked synchronously by client components, *receptacle* identifying an interface required by a client component and provided by another component, *event source* identifying a channel used by a producer component to send events asynchronously, and *event sink* identifying the interface allowing a consumer component to receive asynchronous events. Moreover each component provides a base reference interface offering management and control methods, *i.e.* to configure attributes, to connect facets to receptacles and event sinks to event sources. This management interface is mainly used during deployment and reconfiguration of distributed software component applications.

The different concepts discussed previously, *i.e.* component, reference interface, attribute, synchronous interaction via facet and receptacle ports, and asynchronous interaction via event sink and source ports, are illustrated in Figure 2. These principles are used to design and build both ubiquitous contextual services and our service discovery and deployment infrastructure discussed in Section 4.

In order to build services, it is needed to compose a set of distributed software components. For this purpose, components must be packaged into component archives containing both the component binary implementation and metadata describing context dependencies at least. Moreover, a same component could have different implementations (e.g. a GUI for small PDA screens, another for laptops, etc.). Then a component archive must allow one to store different implementations

[2] Other interaction patterns could be considered as continuous data flow, asynchronous invocations with wait-by-necessity, etc. However these patterns are not currently needed for our prototyped services.

[3] Here we reuse the terminology defined in the CORBA Component Model.

and metadata for each that describes the appropriate running context. Then component archives could be assembled together to form assemblies of distributed software components. These assemblies must identify the component instances to create at deployment time, the configuration of their business attributes, the interconnections between their ports, and finally the physical assignment to nodes. This description could be expressed via an Architecture Definition Language (ADL) or any other metadata formats. Then assembly descriptions and component archives could be packaged together into assembly archives. Then these assembly archives could be deployed automatically as they contain all the required information.

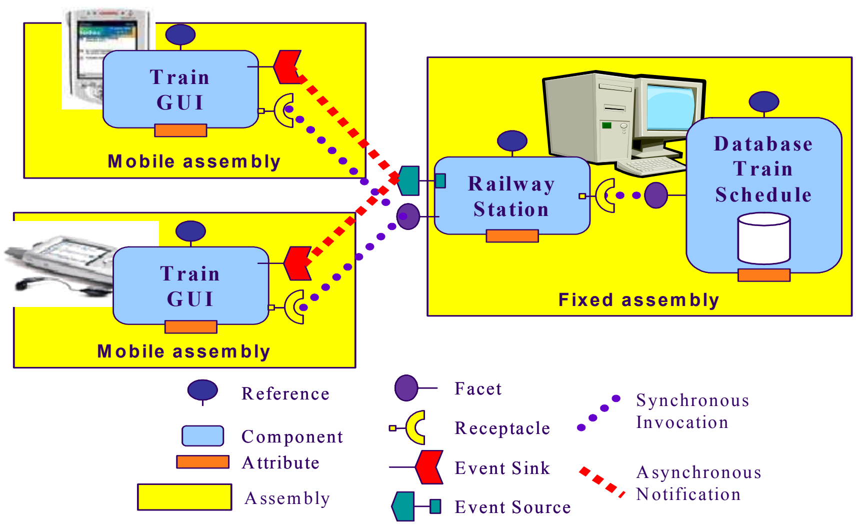

Fig. 2. The train service as a composition of distributed software components.

3.2 The component-based design of the train service

The composition of distributed software components used in our train service is depicted in Figure 2. This service is composed of three main components:

– The Database Train Schedule Component (DTSC) encapsulates the back office information system of the railway operator. It could be deployed on one main frame or replicated over a cluster of servers[4].
– The Railway Station Component (RSC) extracts data related to a specific railway station from the previous component, and it sends them to the user terminals. A railway station component must be deployed on each railway station, it is configured appropriately, *e.g.* with the railway station identifier used to query the back office information system, and it is connected by a receptacle port to the DTSC. This component provides a facet for querying local contextual railway station information, and an event source to notify user terminals of recent updates of this contextual information.

[4] Replication of components is out of the scope of this paper.

– A Train Graphical User Interface (GUI) component presents to end-users the trains which will soon leave from or arrive to the contextual railway station where the user is. For this purpose, this component has two ports - a receptacle and a event sink – to receive data from the contextual TSC, respectively in a synchronous and an asynchronous way. This component is the mobile part of the train service: One instance is deployed and running on each user's mobile terminal. The deployment is done automatically on user's demands. The implementation on the deployed instance depends of the terminal capabilities - e.g. black and white or color screen, and screen size – and user's preferences – e.g. interface language. Then several heterogeneous instances could be simultaneously present in the whole system.

These three components are deployed at different times in the life cycle of the train service. The two first components must be deployed in a preliminary step and will run during the whole life time of the service. On the other hand, Train GUI components are deployed dynamically on user's demands and are instantiated in the system during short periods of times. Then to address this, two different assemblies must be built: One containing the two components running on the fixed computers, and one packaging the mobile part of the service. In order to interconnect Train GUI components to the contextual Railway Station component, it is required that the first assembly exports the RSC reference into a naming or trader registry, and then the second assembly could import the RSC reference from the used registry. In order to allow these incremental deployments and interconnections of assemblies, it is required that the assembly description format provides constructions to express exportations and importations.

4 The Ubiquitous Computing Software Infrastructure

This section describes our software infrastructure dedicated to ubiquitous computing. This infrastructure especially focuses on the problems of service discovery, automatic deployment and execution on users' devices according to their contextual information. We have chosen to model this infrastructure using the distributed component-based approach presented in Section 3. Part of it, *i.e.* a set of components composing the software infrastructure, is dedicated to the discovery of contextual services, whereas some components are responsible of the deployment and the execution of services (*i.e.* the instantiation on the user's device of the client part of the application). These two aspects are discussed in detail respectively in Section 4.1 and 4.2.

4.1 Dynamic discovery of contextual services

The dynamic discovery of contextual services concerns problems raised in Section 2 about the way a user entering a specific area is informed on available services and also if these services are adapted to his/her device. Indeed, considering our example of service providing information on trains, the questions raised are:

– How and when a user arriving in a railway station is informed about the existence of a service providing information on trains, and
– How and when the GUI showing the information could be adapted to the user's device capabilities.

4.1.1 The *Service Registry* and *Service Activator* Components

To address the problem of the discovery of contextual services, we first introduce a component in our software infrastructure named *Service Registry*. The *Service Registry* (*SR*) component is a "server-side" component of the infrastructure architecture. By this we mean that this component will be instantiated on fixed computers of the ubiquitous environment (*e.g.* a server in the railway station). It is in charge of the following main points:

1. Manage the list of available services: the *SR* component maintains a list of services and offers a way for service provider administrators/operators to register, update and remove services.
2. Transmit its unique identifier: every *SR* has a unique ID allowing the differentiation of multiple service providers. Indeed, ubiquitous environments can co-exist from user point of view in some particular cases. Actually, this happens in our example of application if the railway station is near a museum offering ubiquitous services too (like information on current exhibitions, etc.) for instance.
3. Be able to compute the appropriate mobile assembly and/or component implementations of services according to the user's device contextual information (hardware and software capabilities), also called "user profile".
4. Transmit the list of registered services: the list sent to a user should reference only services that are adapted to the device.

Secondly, a component named *Service Activator* represents the "client-side" component of the infrastructure and, as such, has to be started on mobile devices. The *Service Activator* (*SA*) component interacts with the *SR* as follows:

1. Receive *SR* ID: from it, users will be able to access the list of available services registered in the corresponding *SR*.
2. Transmit the user profile: this information is sent by the *SA* component to the *SR* that has been selected by the user.
3. Receive and display a list of available contextual services: from the list, the user can choose and request the deployment of a service.

4.1.2 Discovery of Service Registry and Negotiation Protocol

Thanks to this component-based architecture, the challenge of contextual services discovery can be reformulated as follows: how the *SR* is discovered by the *SA* components? Different scenarios at this stage exist, depending on who - the user/device or the server hosting the *SR* - is initiating the service discovery. Using a protocol based on multicast, the first case, *i.e.* the services search is initiated by user, implies the user's device being in a "transmit" mode (TX), which means to start a periodically sending of search requests. The second case, *i.e.* the user's device is in a "receive" mode (RX), implies that the server is in charge of the periodical sending of requests through the network. This case is more interesting if we take into account several aspects. First of all, considering energy aspect, just let us have a look at power consumption provided by some IEEE 802.11x wireless cards specifications [4, 5, 6]. Table 1 simply shows that the device power consumption is reduced by 25 to 35 percents in the RX mode. This mode will then be implemented in our software infrastructure. Also, this leads us to the following question: which period for requests transmission from server to devices must be used? This period depends on both the mobile device moving speed and the wireless network coverage surface. More the moving speed is important more the period must be short. On the other hand, if the coverage surface is large then the period could be longer.

Table 1. Power consumption of some IEEE 802.11x wireless cards

	TX Mode	RX Mode	RX/TX
802.11b Compaq WL110 [4]	280 mA	180 mA	65 %
802.11b/g Buffalo AirStation G54 [5]	550 mA	350 mA	64 %
802.11a/b/g Dlink DWL-AG660 [6]	500 mA	379 mA	75 %

Next, having one emitter (the service provider server) and several receivers (users' devices) that are waiting for requests generates less network traffic than the opposite. This has to be pointed out in the context of ubiquitous applications as the number of devices involved can be potentially important. At last, the RX mode allows a transparent discovery of the *SR* from users' point of view and updates are automatically and immediately notified to devices.

Once the *SR* is discovered, the user is able to send requests to get the list of contextual services that are adapted to the device. For instance, many implementations of the GUI for the train service are available: GUI for white and black or color screens, text to speech interfaces, resolution, French or English language interfaces, etc. So, depending on the device hardware and software characteristics but also on user's concerns (*e.g.* for the choice of the language of the interface), different component assemblies of the client service part can be proposed. That's the reason why a "negotiation protocol" has been defined between the user's mobile device and the *SR*'s host for the contextual services list discovery. The first step of this protocol concerns the ability of the client service part (the mobile assembly of the service) to be conform to the user/device profile. The hardware/software device information is sent by the *SA* to the *SR* once this last is detected by the user's device. Then, the *SR* computes a list of mobile assemblies that are adapted to the device characteristics and sends it. The *SA* receives this list and the user can then choose among the list of available contextual services displayed. Also, for each service, many assemblies could be available. Indeed, if the *SR* computes that the device can instantiate both color and text to speech interfaces, the user can select between many configurations: color and text to speech interfaces, just the text to speech interface, etc. What's more, other configurations of the service assembly can be considered, not just in term of component implementation choices but also in term of application architecture. To illustrate this point, we can imagine that the user is a *non-French* speaker arriving in a French railway station, so that, if exists, a functional component realizing the *French-to-user's language* translation of the user interface has to be expressed in the alternate configuration.

Finally, all steps of the different interactions between components of our software infrastructure are illustrated in Figure 3, from the *SR* discovery by *SA* (1) to the available contextual services list receiving, including user profile sending (2) and computation of appropriate configuration of mobile assemblies (3).

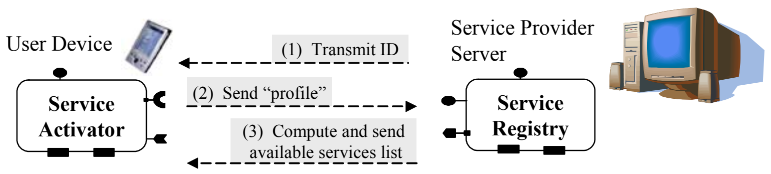

Fig. 3. The negotiation protocol steps of the contextual services list discovery.

4.2 Automatic Deployment and Execution of Services

Once the user has selected a service, the software infrastructure has to deploy the code on the device. In the context of component-based applications, deployment consists of many steps that we can classify in three important phases:

– Component binaries downloading from a repository to devices,
– Instantiation of these components (*e.g.* the GUI) on the user's device, and
– Interconnection them with the components that belong to the fixed part of the service.

For this purpose, we introduce a third component in our architecture that we call *Deployer* (*DP*). As for the *SA* component, it is a "client-side" component and, as such, is instantiated on users' devices. Figure 4 completes our architecture of a software infrastructure for ubiquitous computing and details steps from the service choice by user (4) to deployment of the client part of the application (5, 6a and 6b).

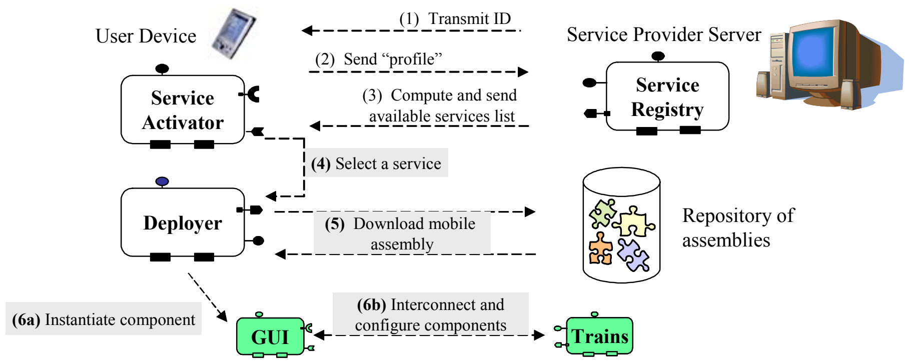

Fig. 4. Service selection and deployment on user's devices.

The components of the service that are running on fixed computers - *e.g.* the component that extracts information on trains to send to users in our example - have been deployed and instantiated in a preliminary step and according to the same deployment process. Assembly that is downloaded on device only contains mobile components to be instantiated on user's terminal and interconnected to previously instantiated fixed components if necessary. The repository of assemblies mentioned in Figure 4 can be located on the service provider server or on a remote HTTP or FTP (or anything else) server.

Once all these steps achieved, end-user can use features provided by the contextual service, that is, in our case, accessing details on trains on departure. Nevertheless, the role of the *DP* component is not limited to deploy and instantiate selected services. In fact, this component is in charge of the entire life cycle of the application. In the context of ubiquitous computing, users arrive and go outside of ubiquitous environments. For instance, the user has taken his train which has started. When users will come in the railway station again, as they know (if they remember) that the "trains" service exists, they would like to use it immediately. That means not having to select one of the displayed *SR* ID, next waiting for the user's profile transmission, choosing the appropriate service and so on. The idea then is to keep in cache mobile assemblies of services that users would be likely to reuse. The *DP* component of our architecture allows that. If the user goes outside the network coverage zone, the *DP* component manages the different scenarios, depending on the type of service:

– The service is available in a "disconnected" state – This case has no consequence for the user (*e.g.* the "mailbox" example).
– The service isn't available in a disconnected state (*e.g.* our "trains" example) – The *DP* asks if the user wants to completely remove the service from device or if he/she wants to keep assembly's archive of the service. In this last case, the mobile assembly is put in a cache (stored in the device flash memory) so that the user will be able to reuse it in future by just instantiating the client part of the application (*i.e.* only from step (6) in our architecture illustrated in Figure 6), obviously if the service is ever available. Service's versions management is done thanks a unique identifier attached to each service.

5 Implementation

To realize our component-based software infrastructure dedicated to ubiquitous computing, we choose the OMG *CORBA Component Model* (CCM). This section previously briefly describes this component model and the OpenCCM platform, which is an open-source implementation of this standard. Next, we detail implementation of the architecture of our software infrastructure defined in Section 4, especially the CORBA components of this architecture and their interactions. Then the implementation of the train service is discussed.

5.1 The OMG CORBA Component Model and OpenCCM

The *CORBA Component Model* is the first vendor neutral open standard for *Distributed Component Computing* supporting various programming languages, operating systems, networks and CORBA products seamlessly [1]. This model is a specification defined by the *Object Management Group* (OMG) consortium for creating distributed, server-side scalable, component-based, language-neutral, transactional, multi-user and secure applications. Software components enhance the object paradigm. Moreover, the CCM defines an architecture dedicated to software engineering and more exactly to the life cycle of software development process: design, implementation, packaging, assembling, deployment, and execution. A CORBA component consists of a reference interface, a set of interaction ports and business attributes. Ports allow one to describe how a component is “communicating” with other components. As previously discussed in Section 3.1, the CCM defines four types of ports: facets, receptacles, event sinks and event sources. Finally, CCM provides an XML-based packaging and assembling facility coupled to an automatic distributed deployment process.

The OpenCCM platform [2] is the first public available and open source implementation of the CCM specification. Since 2002, OpenCCM is hosted by the ObjectWeb consortium, an international consortium promoting open source middleware. The OpenCCM platform is fully written in Java and supports most of available Java-based CORBA products, *i.e.* JacORB, OpenORB, ORBacus or Borland Enterprise Server. OpenCCM is not only a CCM middleware platform for executing CCM applications but it also offers a methodology for building distributed applications including UML-based designing, packaging, assembling, deployment, and management tools. OpenCCM works on various operating systems like Linux, Solaris, Windows NT/2000/XP, and also Windows CE and Linux Familiar for PDA.

5.2 The Software Infrastructure Implementation

First of all, two CORBA components have been defined to respectively model the *SR* and the *SA* components specified in our architecture. The *SR* component provides a facet allowing users to get the list of available services. Also, it is represented by a GUI allowing administrators to manage this list. The *SA* component is represented by a GUI instantiated on user’s device to display, firstly a list of *SR* that have been “detected”, secondly the list of available services registered on each *SR* selected by user. What’s more, a third CORBA component named *Multicast* has been defined to implement the interactions between the *SA* and the *SR* components during the services discovery phase (Figure 5). The generic *Multicast* component provides a facet implementing a “*Send/Receive*” service of multicast requests and is instantiated on each nodes (*i.e.* on all devices and servers). *SA* and *SR* components are both bound to the *Multicast* component (but not the same instance). *SR* uses the “*Send*” service to send its ID whereas *SA* uses the “*Receive*” one to get it. Using of multicast requests is well suited for the implementation of the discovery protocol in the context of a ubiquitous environment. Indeed, *SA* and *SR* components are not directly connected (from a CCM point of view), which was the original goal as user/device is not supposed to know about its environment before discovering it!

As far as the deployment is concerned, a CORBA component implements the *DP* component. It provides a facet used by the *SA* component receptacle. This facet offers many operations for deploying the service. More precisely, the *DP* component is connected, through a receptacle, to the OpenCCM *Distributed Component Infrastructure* (DCI). The DCI infrastructure is designed as a set of CORBA components to manage the deployment phase of CCM applications [7]. Components of the DCI provide facets offering deployment operations as install an assembly by downloading it from a repository to the device, instantiate it or tear down it. Also, the *DP*'s facet implements cache management of assemblies. Finally, the following figure shows the assembly of *SA*, *SR*, *Multicast* and *Deployer* CORBA components, connected to the OpenCCM DCI infrastructure:

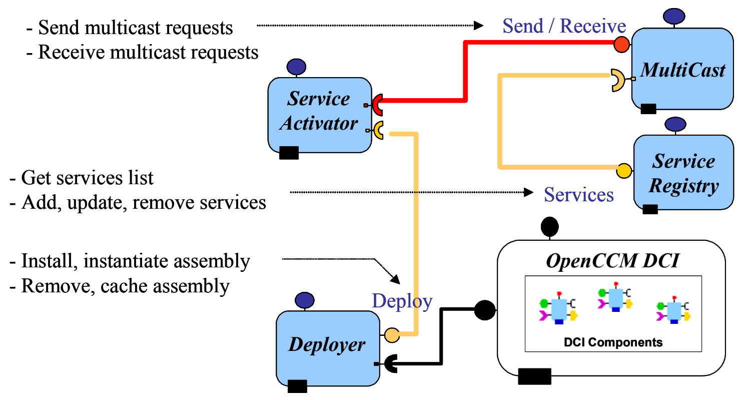

Fig. 5. The CORBA Components Assembly of the Service Discovery and Automatic Deployment Infrastructure.

To experiment our infrastructure, we have also implemented the "trains" service using two CORBA components. A *Server* component is responsible of getting, from service provider's information system (*SP-IS*), and sending, through an event source, information on trains. A *Client* component, deployed on users' devices, receives these data through an event sink and displays a table using a GUI. Figure 6 illustrates on a PDA this example of service at work, *i.e.* after the service's dynamic discovery and its automatic deployment. A laptop has been used to host the *Server* component and the *SP-IS* and the network used is an IEEE 802.11b wireless network.

Fig. 6. An example of service at work on a communicating PDA.

5.3 Limitations

The implementation of our software infrastructure for ubiquitous computing based on the CCM and the OpenCCM platform allows us to address the problems of the contextual services discovery and automatic deployment on user's device according to hardware/software capabilities. Nevertheless, some limitations should be emphasized:

- The size of assembly archives may cause some download time problems during the installation step of the deployment. This is due to the limited throughput of current wireless networks, and the way containers are generated in OpenCCM which implies component binary archives being in some case very large. A dynamic generation of container classes at execution time could address this problem by reducing the size and downloading time of assembly archives.
- The multicast protocol had to be implemented using a specific hand-written component (*Multicast*). This component would be useless if multicast communications between CORBA components will be directly supported by the CCM platform.
- Infrastructure's components (but also many ubiquitous services) use threads. Management of these threads is done in the component business implementation code. This point can be avoided if the component's container will able to manage threads.
- Security during service discovery, automatic deployment, and execution is not currently addressed in our infrastructure. This firstly requires the integration of security mechanisms like SSL secure communications at the ORB level, and access control at the OpenCCM container level. Secondly, security policies dedicated to our infrastructure must be defined and enforced. Some security integration works have been already prototyped for next OpenCCM versions.
- Finally, this approach supposes the client part of the infrastructure (the *SA*, *Multicast* and *DP* components, with the OpenCCM application server and DCI components) being previously installed on devices. Currently, this is done manually. Solutions, as partnerships with device manufacturers for instance, should be considered.

6 Related Works

The OMG/ISO trader [8] and the RM-ODP [9] trader mediate advertisement and discovery of services. The implementation of our service registry could use such technologies.

JINI [10] from Sun Microsystems offers several ways to discover services. The *Multicast Request Protocol* permits to user's terminal to send a multicast request to a lookup service. The response is sent via a TCP unicast message. The opposite solution, named *Multicast Announcement Protocol*, is used by lookup services to announce their presence to any interested parties that may be listening and "in range" of the multicast scope of the lookup service.

The Appear Provisioning Server [11] from AppearNetworks is an infrastructure dedicated to service discovery and deployment. This allows deployment of wireless infrastructure in which applications and documents are available on users' terminal on dedicated places. Applications are developed independently of the infrastructure. In this sense, they do not provide a seamless solution for both infrastructure and services.

The AMPROS project [12] aims to study and develop a middleware for mobile environment. As our proposal, it is based on CCM and the OpenCCM platform. They are developing a set of tools to adapt the applications at runtime.

7 Conclusion

Ubiquitous and context aware applications are one of the new challenges in distributed computing area. Until now, there is no global solution from the design step to the execution step to build such applications. To address this challenge, we have presented in this paper a global solution based on distributed software components to build both ubiquitous context aware applications and the required discovery and automatic deployment infrastructure. This solution has been designed on top of the OMG's CORBA Component Model and implemented using the OpenCCM platform. A sample train service example has been designed, prototyped and already run on top of PDA devices and wireless networks.

Our future works will go in two directions. Firstly, we want to experiment our approach on other service scenarios in order to define a model-driven methodology to build ubiquitous contextual services independently of underlying execution platforms. Secondly, according to the current limitations listed in Section 5.3, we will enhance the OpenCCM platform in several directions: dynamic container generation to reduce the size of assembly archives and improve downloading time on slow wireless networks, integrate multicast communications in the CCM model, integrate a threading service inside CCM containers, and address security at the different required levels (i.e. in underlying CORBA middleware, in runtime containers, and in discovery and deployment components).

8 References

1. Object Management Group: CORBA Components Specification, version 3.0. OMG TC Document formal/2002-06-65. OMG, Boston (2002)
2. OpenCCM Team: The OpenCCM Project. http://openccm.objectweb.org (2002)
3. Szyperski, C.: Component Software: Beyond Object-Oriented Programming (Second Edition). Component Software Series, Addison-Wesley and ACM Press (2002)
4. Compaq: WL 110 Wireless Card User Manual (2001)
5. Buffalo Tech: WLI-CB-G54A User Manual (2002)
6. D-link; DWL-AG660 User Manual (2004)
7. Hoffmann, A., Ritter, T., Reznik, J., Born, M., Neubauer, B., Stoinski, F., Boehme, H., Folliot, B., Vadet, M., Lang, U., Merle, P., Contreras, C.: Specification of the Deployment

and Configuration. IST COACH Document Deliverable #2.4. http://www.ist-coach.org (2003)

8. Object Management Group: Trading Object Service Specification, verion 1.0. OMG TC Document formal/2000-06-27. OMG, Boston (2000)
9. Leydekkers, P.: Multimedia Services in Open Distributed Telecommunications Environments. PhD Thesis CTIT (1997)
10. Edwards, W. K.: Core Jini Introduction. Prentice Hall PTR (1999)
11. AppearNetworks: The Appear Provisioning Server. http://www.appearnetworks.com
12. Ayed, D., Taconet, C., Bernard, G.: Deployment and Reconfiguration of Component-based Applications in AMPROS. Proactive computing workshop (PROW 2004) - Helsinki, Finland (2004)