An Architecture for Ubiquitous Applications

Sofia Zaidenberg[1], Patrick Reignier[1], James L. Crowley[1]

[1] Laboratoire LIG
681 rue de la Passerelle - Domaine Universitaire - BP 72
38402 St Martin d'Hères
{Zaidenberg, Reignier, Crowley}@inrialpes.fr
http://www-prima.imag.fr

Abstract. This paper proposes a framework intended to help developers to create ubiquitous applications. We argue that context is a key concept in ubiquitous computing and that, by nature, a ubiquitous application is *distributed* and needs to be *easily deployable*. Thus we propose an easy way to build applications made of numerous modules spread in the environment and interconnected. This network of modules forms a permanently running system. The control (installation, update, etc.) of such a module is obtained by a simple, possibly remote, command and without requiring to stop the whole system. We ourselves used this architecture to create a ubiquitous application, which we present here as an illustration.

1. Introduction

New technologies bring a multiplicity of new possibilities for users to work with computers. Not only are spaces more and more equipped with stationary computers or notebooks, but more and more users carry mobile devices with them (smart phones, PDAs, etc.). Ubiquitous computing takes advantage of this observation. Its aim is to create smart environments where devices are dynamically linked in order to provide new services to users and new human-machine interaction possibilities. *The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it* [20]. A ubiquitous environment is typically equipped with sensors, for example cameras, Bluetooth dongles, WiFi routers, RF-Id readers, etc. Users are detected using vision or using the mobile devices they carry, for instance Bluetooth enabled cell phones, PDAs with WiFi cards, RF-Id tags built in their professional badges, etc. Such an environment should also provide numerous human-machine interfaces, as for example screens and touch-screens, microphones for speech recognition, speakers for speech synthesis, keyboards and mice or more sophisticated input-output devices (for instance [2]). All these devices, by definition of ubiquitous, are spread in the environment. Moreover, they can be mobile, appearing and disappearing freely as users come and go. As a consequence, a ubiquitous system is necessarily *distributed*. We are in an environment where computers and devices are numerous and spread out and need to cooperate in order to achieve a common goal. Moreover, the environment should not put

constraints on the kind of platforms it integrates. In fact, legacy devices are already present in the environment and the ubiquitous system should use them. Additionally, if the system integrates the users' mobile devices, it can not impose a certain kind of device because users would not accept such a constraint.
To come to the point, in order to achieve its vision, ubiquitous computing must (i) Integrate numerous and heterogeneous platforms that are spread out and that can dynamically appear and disappear. (ii) Perceive the user context in order to enable implicit interaction.
This is the background when developing ubiquitous applications. What we propose is an implementation of this background, of the software infrastructure underlying a ubiquitous system. Our architecture comprises the communication and interconnection of modules, service discovery, a platform for easy deployment, dynamic recomposition without restart. This architecture is easy to install on every device of the environment, it works on Linux, Windows and MacOSX. Modules are easy to develop thanks to a wizard (section 3.2.3) for Java. Modules can also be written in C++ and can communicate with the others. Thanks to this architecture, developers can focus on the algorithms which make the intelligence of the system and address fundamental problems. The cost of adding a module and deploying it for testing on several machines is minimal. One can install, uninstall, start, stop or update a module without suspending the whole system.
Furthermore, we propose a centralized knowledge base. This database records the history of modules' lifecycles, of actions and events in the environment. It also contains static information about the infrastructure and the users. This database helps the development of ubiquitous applications because it allows to replay a scenario, to find out everything that happened during an experience, etc.

2. Related Work

Context has been recognized as being a key concept for ubiquitous applications [4], [9]. Dey [4] defines context to be "*any information that can be used to characterize the situation of an entity, where an entity can be a person, place, or physical or computational object*". A number of frameworks have been proposed to facilitate context-awareness in ubiquitous environments. Some of them are centralized context servers, for instance Schilit's mobile application customization system [16], the Contextual Information Service (CIS) [13], the Trivial Context System (TCoS) [7] and the Secure Context Service (SCS) [1], [10]. These systems operate as middleware collecting unprocessed data from sensors, treating it and providing interpreted, high level context information to applications. In this manner it is easier to ensure data integrity, aggregation and enrichment. On the other hand it is harder to make such a system evolve in size and complexity. Most of this work is rather early in ubiquitous computing. Distributed systems have been proposed, as for example the Context Toolkit [4] or the architecture proposed by Spreitzer and Theimer [17], where context information is broadcasted. The Context Toolkit offers distributed context components and aggregators that make the distributed character of the environment transparent to the programmer.

More recent work proposes frameworks for ubiquitous applications, helping developers to use context: Rey [3] introduces the notion of a *contextor*, a computational abstraction for modeling and computing contextual information. A contextor models a relation between variables of the Observed System Context. An application is obtained by composing contextors. This work is an extension to the Context Toolkit as it adds the notion of metadata and control to the system.
Our work relates to these frameworks as it also proposes a structure for developing ubiquitous applications. But our architecture is more general. In fact, we propose no more than an easy way to create a number of interconnected modules that are distributed in the environment and whose lifecycles can be remotely controlled. By "easy way" we intend to say that the devise of one module and the deployment of the global system are simple. Our architecture stays in the realm of ubiquitous computing as it is oriented for interactive applications: it has a small latency and a small network cost.
In the following sections we describe our architecture. We state the needs that appear when building ubiquitous applications and that we respond to. We then explain how we respond to those needs and what the underlying technologies that we use are. Finally we illustrate our work by an example of a ubiquitous application developed based on our architecture.

3. An architecture for ubiquitous applications

3.1. Needs

As we described above (section 1), a ubiquitous environment needs to integrate heterogeneous platforms: computers running with different operating systems (Linux, Windows, MacOSX, etc.) and mobile devices such as smart phones or PDAs (which can also have various operating systems). Therefore, a ubiquitous computing architecture is *distributed*. This implies the need for a communication protocol between the modules that are spread in the environment. Furthermore, it implies the need for a dynamic service discovery mechanism. In fact when a module needs to use another module, for instance a speech synthesizer, it should dispose of an easy way to find it. The knowledge of the host it is currently running on should not be necessary. Moreover, often the need would be to find a speech synthesizer in a given location (the location of the user). Thus we first need a way to dynamically find a host with the additional constraint that it is connected to speakers located in that room; and then find the needed module on that host. It would be even better if we could dynamically install and/or start this module if it is not already available. More generally, it is convenient to be able to control the deployment and the lifecycles of modules. This example shows that we also need a knowledge base on the infrastructure and the registered users. This component knows that there are speakers in the given room and it knows the hostname of the device connected to them. With the perspective of providing services to users, this knowledge should include user preferences such as the preferred modality for interaction (e.g. audio rather than video) or the actions to be

executed in a certain situation (e.g. turn on the lights when the luminance is lower than a threshold). This knowledge could be spread: each machine knows its hardware and capabilities. But we made the choice of a centralized database with a “map” of the environment in terms of hardware. In fact, we are going to query this knowledge base very frequently and it is more convenient to group all this information together, rather than search on several hosts for the answer.
In addition, for more flexibility and a better scalability, we should admit that these modules can be written in different programming languages. Such a distributed architecture can easily become hard to maintain. In fact, installing software on several heterogeneous machines is work. Keeping this software up-to-date is even more laborious. The update should be easy and quick and it should not require the stop of the whole system, or even the stop of the machine being updated.
We propose such an architecture, responding to the needs mentioned above with the combination of two recent technologies: OMiSCID [6] for communication and service discovery and OSGi [12] for deployment. Additionally, we designed a database serving as the central knowledge component. We use a context model to tell us what to do in each situation for each user. This constitutes a higher level layer. The following sections detail these aspects.

3.2. Architecture

In this section we describe in detail which technologies our architecture is based on and how one uses it to create ubiquitous applications.

3.2.1. Communication between modules: OMiSCID

As shown above (section 3.1), we need a communication protocol between modules. For more flexibility and scalability, each module is an independent piece of software and can be devised by different developers. Such a communication protocol has been specified and implemented by [6]: OMiSCID is the resulting middleware. OMiSCID is composed of two layers: (i) network communication: BIP [11]. The network cost is minimal: only a 38 bytes heading per message. (ii) The service layer. It uses DNS-SD [5], also called Bonjour (c.f. Apple) as a distributed directory. Each module is implemented as an OMiSCID “service”. At startup, it declares itself to the domain and can be contacted by any other module in the same domain, whether they are physically running on the same machine or not. This contact is made through “connectors” defined by each module. Three types of connectors exist: input, output and inoutput. There are several ways to make modules communicate. A sensor module can have an output connector on which it sends messages when it detects events. For instance a person tracker could send an event each time a person enters or leaves the area; a luminance sensor could fire events when the luminance goes past some threshold. A client module connects an input connector with the output connector of the sensor to receive these events (and react to them using listeners). To send a command to an effector module, the client connects its output connector with an input connector of the effector and uses it to send a request. Two modules can also communicate through two connected inoutput connectors. The client sends a request to the server, which answers it through the same connector. Figure 1 shows the connectors of a personal assistant (called PersonalAgent), which is described section 3.3 below. “-” represent inoutput connectors, “+” – input and “#” – output connectors. An OMiSCID services also defines variables representing its state. Variables can have read or read-write permissions.

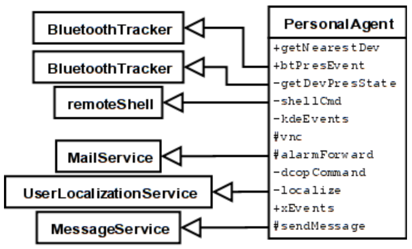


Figure 1: The personal assistant and his connectors. Each connector is used for communication with a particular OMiSCID service. (Not all connections are shown.)

This architecture allows us to connect different modules, developed possibly by different programmers, and to use each module like a black box, knowing only the message convention for communication (see section 3.2.1.1 below). Moreover, this facilitates the ubiquitous deployment of the system. In fact, modules can run on different hosts, including mobile devices: it is possible to implant OMiSCID and OMiSCID modules on a PDA connected to the network via WiFi. Additionally, OMiSCID is multi-language and multi-platform as it is implemented in C++ for Windows, Linux and MacOSX, and in Java. In this matter, ubiquitous applications may runs on one machine, but are ubiquitous because they have very easy and transparent access to any other mobile or stationary device of the environment, “equipped” with a module providing access to it.

3.2.1.1. Communication format

For modules to understand each other, we need a convention for the format of exchanged messages. We made the choice of the XML format for messages. Each connector is associated to an XSD schema (which can be retrieved from the database) and incoming or outgoing messages are valid XML strings for the corresponding schema. To facilitate the processing of this XML, we use Castor [21] for a mapping between XML and Java objects. In this manner, developers work only with objects. They do not need to know exactly the syntax of messages, but only the semantic (which should be easy to infer from the fields of the Java objects).

3.2.2. Deployment of modules: OSGi

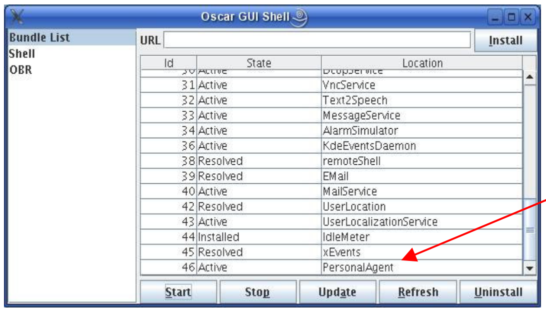
At this point we have a distributed architecture of inter-connected modules. Such architecture can quickly become hard to control and to maintain. For instance, a person tracker would run on several machines in order to monitor several rooms. We need an easy way to install and update the person tracker on all these machines when a new version is developed.
Furthermore, we are in an environment with numerous stationary machines, and where devices can appear dynamically. We can not consider installing every module manually on every machine. We need an opportunistic strategy: we install a module on a machine only when needed. For instance we install a speech synthesizer on a

machine in a room when we need to send a message to a user in that room. The dynamic deployment of modules is based on a central server containing all available modules.
OSGi [12] provides these functionalities. We use Oscar as an implementation of OSGi. On each device that belongs to the ubiquitous environment, an Oscar platform runs. At least two bundles are required: one that incorporates jOMiSCID, a Java implementation of OMiSCID[1] and one that incorporates Castor, allowing modules to decode messages they send and receive (section 3.2.1.1). All the modules are at the same time OSGi bundles and OMiSCID services. A common bundle repository is created on a central machine and it regroups all the modules. It is accessed via http. Every Oscar platform installs its bundles from this repository and thus is linked to it. When a new module is developed, it is added to the repository and all the Oscar platforms can just install the corresponding bundle. When a module is updated in the repository, all the Oscar platforms just have to update their bundle. This is very convenient because once this architecture is set up, new modules are very easy to install. One doesn't need to compile them on the new machine or to worry about dependencies. To update a module, one also has to just enter the "update" command in Oscar.
It is also convenient to be able to remotely control other modules (start a needed module, update, stop or even install one). We add this functionality to OSGi via a bundle called "remoteShell". It is an OMiSCID service able send commands to the Oscar platform that it runs on. Thus a distant module can call upon the local "remoteShell" (through OMiSCID connectors) and send OSGi commands to it. This bundle should be installed by default on every Oscar platform as well.

3.2.3. For easy development: a wizard

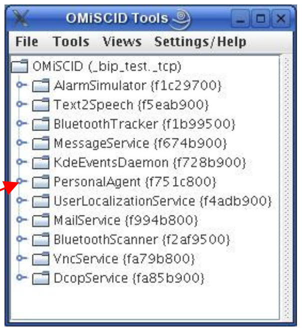
To facilitate the programmer's work, we have developed an Eclipse plug-in [14]. It provides a wizard for creating new projects of the type OMiSCID service and OSGi bundle. An empty project generated with this wizard contains the correct arborescence of packages and files, the main classes with the OMiSCID code to create and register the service, the XML file where the developer should specify the connectors, the lifecycle code for OSGi, the manifest file and the build files ready to build and publish the new bundle on the repository. Thus to add a new empty module, one (i) uses the wizard, (ii) builds and publishes the new project by running the generated `build.publish.xml` file (iii) installs it in Oscar (a single `install` command) and (iv) starts it in Oscar (a single `start` command). At this point the new OMiSCID service is visible in the domain.
A user interface has been created to visualize the services running in a domain. It is shown Figure 3. Figure 2 shows the graphical interface of Oscar.

[1] The Java version of OMiSCID – jOMiSCID – is not a JNI layer of the C++ version, but a complete rewriting in pure Java.



Id	State	Location
30	Active	DcopService
31	Active	VncService
32	Active	Text2Speech
33	Active	MessageService
34	Active	AlarmSimulator
36	Active	KdeEventsDaemon
38	Resolved	remoteShell
39	Resolved	EMail
40	Active	MailService
42	Resolved	UserLocation
43	Active	UserLocalizationService
44	Installed	IdleMeter
45	Resolved	xEvents
46	Active	PersonalAgent

Figure 2: The graphical interface of Oscar, showing the bundles. Notice that the active bundles are visible as services on Figure 3.



```
OMiSCID (_bip_test._tcp)
  AlarmSimulator {f1c29700}
  Text2Speech {f5eab900}
  BluetoothTracker {f1b99500}
  MessageService {f674b900}
  KdeEventsDaemon {f728b900}
  PersonalAgent {f751c800}
  UserLocalizationService {f4adb900}
  MailService {f994b800}
  BluetoothScanner {f2af9500}
  VncService {fa79b800}
  DcopService {fa85b900}
```

Figure 3: Visualization of the OMiSCID services running in a domain.

3.3. Example: a personal ubiquitous assistant

We use the architecture described above (section 3.2) for a personal ubiquitous assistant. Our environment is equipped with a number of devices which we use to retrieve information about the user and his context. Mobile devices that users bring into the environment contribute to that knowledge. We use that knowledge to propose relevant services to the user. The assistant is the central module of the environment, observing the user and offering him proactive services. The assistant can do task migration: doing a task instead of the user. It can also provide additional services, for instance forwarding a reminder to the user if he receives it while being away from his computer. Thus our assistant is personal in the sense that it is user-centered. It is ubiquitous because it uses information provided by all available devices in the environment to observe the user and estimate his current situation or activity. It calls upon these devices to provide services as well. Thereby, the services provided are *context-aware*. At last, our assistant is a learning assistant because these services are *user-adapted*. In fact, most of current work on pervasive computing [19], [15] pre-defines a number of services and fires them in the correct situation. Our assistant starts with this pre-defined set of actions and adapts it progressively to his particular user. The default behavior allows the system to be ready-to-use and the learning is a life-long process, integrated into the normal functioning of the assistant. Thus, the assistant will, at first, be only acceptable to the user, and will, as time passes, give more and more satisfying results.
The assistant observes the user and gathers clues on his context and his activity. For that he needs perceptive modules – sensors – able to provide this kind of information. For providing services to the user, it needs proactive modules – effectors. Some modules can run on any host and some are linked to, or used to access, specific hardware. Thus they run on the machine connected to this hardware. For instance, a person tracker needs at least one camera. The person tracker module is then installed on the machine connected to the camera(s). In the same way, each machine equipped with speakers would have a module able to synthesize speech or play given audio input.

To sum up, our ubiquitous system is composed of several modules and the personal assistant. Sensor modules can fire events, received by the assistant, or the assistant can explicitly interrogate a sensor. This input allows the assistant to estimate the user's situation (section 3.3.1.1). The default behavior, possibly modified by acquired experience, indicates to the assistant how to act in a certain situation. When the appropriate action is chosen, the assistant contacts an effector to execute it (e.g. to provide the chosen service).

3.3.1. Mechanism of the assistant

We mentioned in section 3.3 above an example of a service provided by the assistant. In this example, the user is currently away from his office and his agenda fires a reminder. The system detects this event. A rule indicates it how to react in this situation to this event. The assistant now knows that it must inform the user of the reminder. It tries to find the user in the building and to send him a message with the reminder. When contacting the user, the assistant takes into account the context of the user (whether he is alone or not) and the preferences of the user (his preferred modality for receiving a message). In the following sections, we explain in detail how this example works.

3.3.1.1. The context model

A context is represented by a network of situations [8]. A situation refers to a particular state of the environment and is defined by a configuration of entities, roles and relations. An entity is a physical object or a person, associated with a set of properties. It can play a role if it passes a role acceptance test on its properties. A relation is a semantic predicate function on several entities. The roles and relations are chosen for their relevance to the task. Likewise, only entities that can possibly play roles and that may be relevant for the task are considered. A situation represents a particular assignment of entities to roles completed by a set of relations between entities. Situation may be seen as the "state" of the user with respect to his task. If the relations between entities change, or if the binding of entities to roles changes, then the situation within the context has changed. To detect situation changes, a federation of observational processes is required.

In order to provide services, one can define rules that attach actions to situations. These actions are triggered when the situation the system finds itself in, changes. Corresponding services are then provided to the entities (users) playing the required roles. Figure 4 shows an example of a context model. This context model tells the assistant what to do in a given situation and what the current situation is given the observations of the user and the environment. The context model incorporates the intelligence of the assistant.

This context model is implemented in Jess as a Petri network.

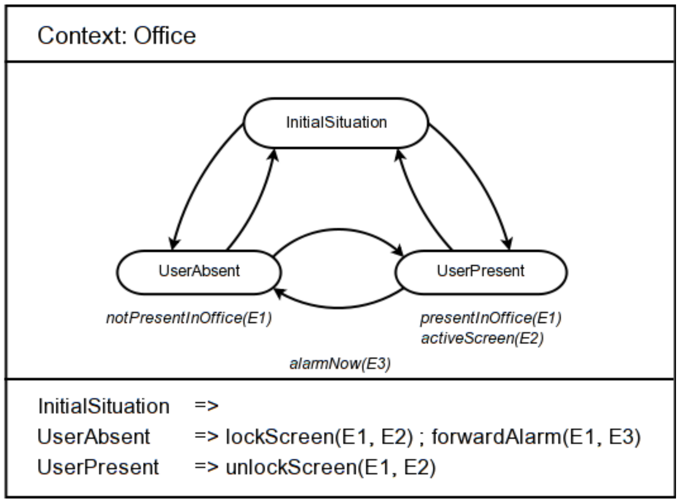


Figure 4: Extract of the context model

3.3.1.2. The ubiquitous database

All modules share a database divided into four parts (*user*, *service*, *history* and *infra-structure*) and partly presented Figure 5. Each part is implemented as an SQL schema.

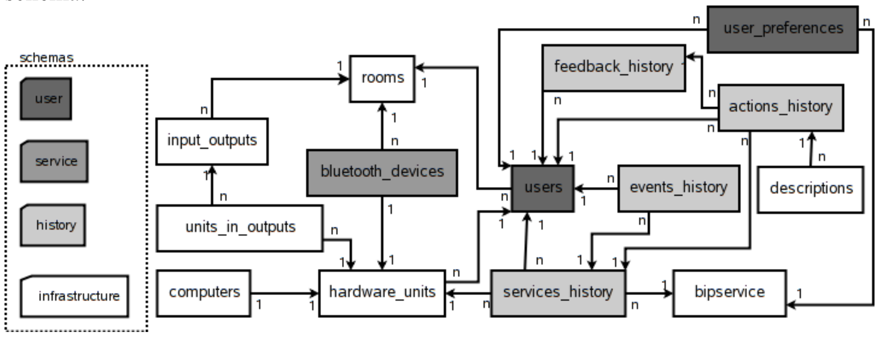


Figure 5: Simplified scheme of the database

The part *history* stores the modules lifecycles (the dynamic start and stop of bundles in the OSGi platform), all occurred events and actions taken by the system. This part is useful for explaining to the user (if required) why an action was or was not taken. We believe that these explanations bring a better trust of the user for the intelligent system.
The part *infrastructure* contains known, static information about the environment: the rooms of the building and their equipment (in- and output devices and the hardware units that can control them). This knowledge can be used to determine which device

is the most appropriate for the needs of a service. The part *user* describes registered users (it associates users with their Bluetooth devices and allows to identify users; it stores user logins allowing to identify them on the local network, etc.). As a future evolution, this user data will be brought in by the user on his PDA as a profile. When the user enters the ubiquitous environment, his profile is loaded into the system. This discharges the PDA of any heavy treatment that could be necessary to exploit this personal data (learning algorithms, HMMs, etc.). The computation is taken over by another device: a central server, the personal computer of the user, etc. and the PDA, whose resources are very limited, stays available for its primary purpose. When the user leaves the environment, it is conceivable to migrate the new profile back on the PDA. In this way, the user has the updated data with him if he needs it elsewhere (in another ubiquitous environment, for instance at home or in his car or maybe in a public place).
The part *service* is a free ground for plug-in services that are not necessary for the core system to function. For instance if we dispose of Bluetooth USB adapters, we can use them to detect users' presence by detecting their Bluetooth cell phones and PDAs. This information may be used to determine the identity of a video tracker's target.
In order to easily communicate with the database, we use Hibernate [18]. We enclose it into a bundle that other modules use to query the database. We implement all the needed queries in this bundle and other modules just call functions to execute them. This makes modules more independent of the database. If the database is modified, only the one specific bundle has to be updated.
The database is the knowledge and the memory of the assistant. It tells it where to send a command to provide a service. It can also be used to explain actions of the assistant to the user. If the preferred modality was not chosen, we can explain using the database that it was not available in the given location. As the database keeps a history of all events and actions, and of the lifecycle of all modules, it can explain, for instance, in case of a failure, that it happened because a certain module was not running at the time.

3.3.1.3. The global mechanism

Figure 6 resumes the logic of our system and shows the connections between components. The personal assistant is the central part of the system. Its role is to connect all the resources and knowledge of the environment in order to reach the goal of providing services to the user. It is linked to the environment by the modules mentioned above (section 3.2). The latter run in the OSGi platform as bundles and can communicate with each other (and with the assistant who is such a module as well) through OMiSCID (as they also are OMiSCID services).
Perception gives the assistant information about the user, for instance his location and his activity. These clues are directly interpreted by the assistant as the *role* (in the sense defined above, section 3.3.1.1) the user is playing. It sends it as an input to the context model. That is to say, it asserts Jess facts which possibly trigger rules that make the context model fire a transition and thus switch the current situation. The activation of a situation triggers also rules defining services that are to be provided to

the user. Concretely, Jess rules call methods of the assistant who knows how to provide that service and which OMiSCID services (effector modules) to contact.
The ubiquitous database is used all along in this process. It is mainly queried by the assistant, but also by other modules.

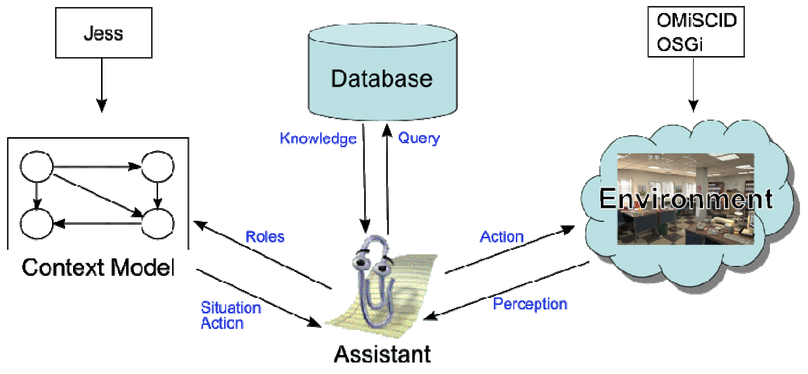


Figure 6: The global mechanism of the system

Let us reconsider the example scenario mentioned section 3.3.1 above in order to explain in depth how it works. The user's agenda (KOrganizer) triggers a reminder. A module ("KAlarmDaemon") was listening to reminders thus it detects this event. It sends a message on his output connector. The assistant has an input connector connected to the output of "KAlarmDaemon" thus it receives this event. The alarm is considered as an *entity* playing the role "AlarmNow" hence the assistant asserts the corresponding Jess fact. If the current situation is "UserPresent", nothing else happens. But let us consider the user being absent and therefore the current situation being "UserAbsent". In this situation, a rule indicates that the reminder should be forwarded to the user. Actually, there are three rules differentiating the cases where the user location is unknown and where the user is in a known office but alone or not alone[2]. Thus one of these rules is fired by the Jess engine and calls a method of the assistant. The context model (implemented as Jess rules) knows what to do and the assistant knows how to do it. In the case where the user is not to be found, the only way to forward the reminder is by email[3]. Thus the assistant contacts the email module "MailService" (the user's email address is known from the database) which sends an email using JavaMail. If the user location is known, the assistant tries to contact the user by his preferred modality (written message, synthesized voice, etc.), which is of course different whether the user is alone or not. Thus, for each modality in preference order, the assistant queries the database to find out if it can be provided in the room the user is in. When it obtains a positive answer, it queries the database for the machine (hardware unit) connected to the correct output (screen, speaker, etc.). Then it searches on that host for the corresponding module capable of providing that ser-

[2] The user location is maintained up to date in the Jess engine by the assistant who is connected to a module sending out user location events.

[3] Of course, a text message on the user's mobile phone would be even more adequate, but at the moment we do not have the technical means to do that.

vice. If the corresponding OSGi bundle is currently not started, it tries to remotely start it (or even install it first) and then to contact it. One can notice that in this example, the host that is contacted to forward the reminder could very well be a mobile device, for instance the user's PDA, as long as it is in a WiFi covered area and was equipped with OMiSCID and Oscar.

3.3.1.4. The interconnected modules used by the assistant

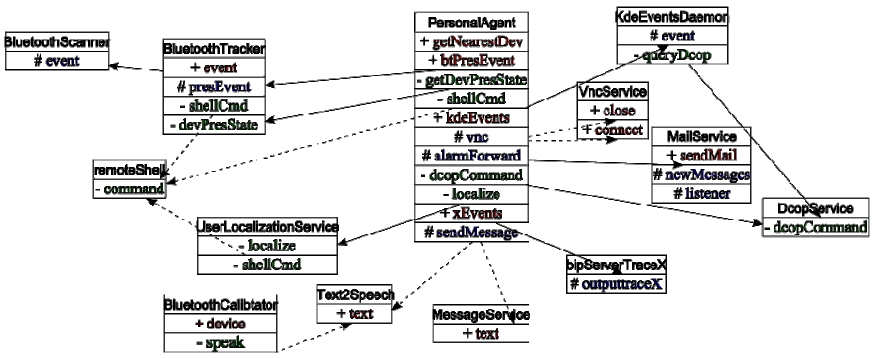


Figure 7: A schema of our modules and their interconnections. Input connectors are marked with "+", output with "#" and inoutput with "-".

Figure 7 shows our modules and the way that they communicate. The assistant corresponds to the module "PersonalAgent". Modules are OMiSCID services and have input ("+"), output ("#") and inoutput ("-") connectors. An arrow from connector A to connector B means that A makes a connection to B (A needs B). It does not indicate the direction of messages (for instance if an input connector connects to an output connector, the messages go in the opposite direction: from the output to the input). The dotted arrows indicate that the connection is only made if needed, dynamically during the execution. The plain arrows stand for permanent connections made at start-up. When such a connection breaks, the module that needs it waits for the other module to return (or tries to restart it). This wait blocks the module if the connection is necessary or it can be done in background if the information provided by the connection is optional. One can notice that two modules can be linked by several connections: for a better clarity of the system, one connection should correspond to one semantic need.

4. Conclusion

We presented in this paper a framework for ubiquitous applications. Our aim is to provide developers an easy way to build such systems. Looking at the requirements of ubiquitous computing, we created an adapted software architecture providing simple and convenient means to create ubiquitous software modules. The interconnection and communication of these modules forms the ubiquitous application. As we provide technical solutions to easily deploy and interconnect modules, developers can focus on algorithms that will constitute the core intelligence of their systems. We use OMiSCID for service discovery and communications. OSGi provides us a framework for deployment of modules without stopping the whole system. The combination of OMiSCID and OSGi allows us to add the remote control of modules' lifecycles.

5. References

[1] Bisdikian, Chatschik and Christensen, Jim and Davis, John II and Ebling, Maria R. and Hunt, Guerney and Jerome, William and Lei, Hui and Maes, Stéphane and Sow, Daby. Enabling location-based applications. In *WMC '01: Proceedings of the 1st international workshop on Mobile commerce*, pages 38-42, New York, NY, USA, 2001. ACM Press.

[2] Borkowski, Stanislaw and Letessier, Julien and Crowley, James L. Spatial Control of Interactive Surfaces in an Augmented Environment. In Remi Bastide and Philippe A. Palanque and Jorg Roth, editor, *Engineering Human Computer Interaction and Interactive Systems, Joint Working Conferences EHCI-DSVIS 2004, Revised Selected Papers*, volume 3425 of Lecture Notes in Computer Science, pages 228-244. Springer, July 2004. EHCI 04.

[3] Coutaz, Joëlle and Rey, Gäetan. Foundations for a Theory of Contextors. In *CADUI*, pages 13-34, 2002.

[4] Dey, Anind K. and Abowd, Gregory D. The Context Toolkit: Aiding the Development of Context-Aware Applications. In *The Workshop on Software Engineering for Wearable and Pervasive Computing*, Limerick, Ireland, June 2000.

[5] Dns-sd. Dns service discovery. http://www.dns-sd.org.

[6] Emonet, Rémi and Vaufreydaz, Dominique and Reignier, Patrick and Letessier, Julien. O3MiSCID: an Object Oriented Opensource Middleware for Service Connection, Introspection and Discovery. In *1st IEEE International Workshop on Services Integration in Pervasive Environments*, June 2006.

[7] Hohl, Fritz and Mehrmann, Lars and Hamdan Amen. *Trends in Network and Pervasive Computing - ARCS 2002: International Conference on Architecture of Computing Systems, Karlsruhe, Germany, April 8-12, 2002. Proceedings, volume Volume 2299/2002 of Lecture Notes in Computer Science*, chapter A Context System for a Mobile Service Platform, page 21. Springer Berlin / Heidelberg, February 2004.

[8] J. L. Crowley and J. Coutaz and G. Rey and P. Reigner. Perceptual components for context awareness. In *International conference on ubiquitous computing*, pages 117-134. Springer Verlag, 2002.

[9] Jakob E. Bardram. *Pervasive Computing*, volume 3468/2005 of *Lecture Notes in Computer Science*, chapter The Java Context Awareness Framework (JCAF) - A Service Infrastructure and Programming Framework for Context-Aware Applications, pages 98-115. Springer Berlin / Heidelberg, May 2005.

[10] Lei, Hui and Sow, Daby M. and Davis, John S. II and Banavar, Guruduth and Ebling, Maria R. The design and applications of a context service. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):45-55, 2002.

[11] Letessier, Julien and Vaufreydaz, Dominique. Draft spec: BIP/1.0 - A Basic Interconnection Protocol for Event Flow Services. http://www-prima.imag.fr/prima/pub/Publications/2005/LV05/, 2005.

[12] OSGi. Open Services Gateway initiative. http://www.osgi.org.

[13] Pascoe, Mr. Jason. Adding Generic Contextual Capabilities to Wearable Computers. *iswc*, 00:92, 1998.

[14] Patrick Reignier. Plugin Eclipse pour la création d'un projet OMiSCID. http://www-prima.inrialpes.fr/reignier/update-site/.

[15] Ricquebourg V., Menga D., Durand D. , Marhic B., Delahoche L. and Log C. The Smart Home Concept: our immediate future. In IEEE Industrial Electronics Society, editor, *Proceedings of the First International Conference on E-Learning in Industrial Electronics*, Hammamet - Tunisia, December 2006. ICELIE'06.

[16] Schilit, Bill N. and Theimer, Marvin M. and Welch, Brent B. Customizing Mobile Application. In *USENIX Symposium on Mobile and Location-independent Computing*, pages 129-138, Cambridge, MA, US, August 1993.

[17] Spreitzer, Mike and Theimer, Marvin. Providing location information in a ubiquitous computing environment (panel session). In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 270-283, New York, NY, USA, 1993. ACM Press.

[18] Steve Ebersole. Hibernate Core for Java. http://www.hibernate.org/.

[19] Vallée, M., Ramparany, F. and Vercouter, L. Dynamic Service Composition in Ambient Intelligence Environments: a Multi-Agent Approach. In *Proceeding of the First European Young Researcher Workshop on Service-Oriented Computing*, Leicester, UK, April 2005.

[20] Weiser, Mark. The computer for the 21st century. *Scientific American*, 265(3):66-75, September 1991.

[21] Werner Guttmann. The Castor Project. http://www.castor.org/.