An Architecture for Ubiquitous Applications

Sofia Zaidenberg, Patrick Reignier and James L. Crowley

Laboratoire LIG 681 rue de la Passerelle - Domaine Universitaire - BP 72
38402 St Martin d'Hères

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* 16. A ubiquitous environment is typically equipped with sensors. Users are detected using vision or the mobile devices they carry. Such an environment should also provide numerous human-machine interfaces. Moreover, all the devices in the environment 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, by definition of ubiquitous, and need to cooperate in order to achieve a common goal. Furthermore, the environment should not put constraints on the kind of platforms it integrates. In fact, legacy devices are already present and should be used. Additionally, the system integrates users' mobile devices, and thus can not put any constraints on them.

To come to the point, in order to achieve its vision, ubiquitous computing must (i) Integrate numerous, heterogeneous and spread out platforms that can dynamically appear and disappear. (ii) Perceive the user context 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 0) 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 by allowing to replay a scenario, to examine an experience, etc.

2 Related Work

Context has been recognized as being a key concept for ubiquitous applications 3, 7. Dey 3 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 are centralized context servers, such as Schilit's mobile application customization system 13, the Contextual Information Service (CIS) 10, the Trivial Context System (TCoS) 5 and the Secure Context Service (SCS) 1, 8. 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 3 or the architecture proposed by Spreitzer and Theimer 14, 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 2 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 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, distributed modules whose lifecycles can be remotely controlled. 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 0), a ubiquitous environment needs to integrate heterogeneous platforms: computers running with different operating systems and mobile devices such as smart phones or PDAs. 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, *e.g.* a speech synthesizer, it needs to find it in the environment first. Moreover, often the need would be to find a speech synthesizer in the location of the user. Thus we first need to dynamically find a host 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). 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, queries of this knowledge base are very frequent and it is more convenient to group the information, rather than search on several hosts.

In addition, for more flexibility and a better scalability, modules can be written in different programming languages. Such a distributed architecture is hard to maintain. In fact, installing software on several heterogeneous machines and keeping it up-to-date is laborious. The update should be easy and quick, without requiring the stop of the whole system, or even the stop of the machine being updated.

We propose such an architecture, responding to these needs with the combination of two recent technologies: OMiSCID 4 for communication and service discovery and OSGi (www.osgi.org) for deployment. Additionally, we designed a database serving as the central knowledge component. 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.

Communication between Modules: OMiSCID. As shown above (section 0), 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 protocol has been specified and implemented by 4: OMiSCID is the resulting middleware, composed of two layers: (i) network communication: BIP 9 with a minimal network cost: only a 38 bytes heading per message. (ii) The service layer using DNS-SD (www.dns-sd.org), 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 service in this domain, whether they are physically running on the same host or not. This contact is made through "connectors" defined by each module. Three types of connectors exist: input, output and inoutput. Modules communicate in several ways. An output connector can broadcast messages. For instance a person tracker sending an event each time a person enters or leaves the area. To receive messages (and react to them using listeners), a module connects its input connector to an output connector. To send a message and receive an answer, a module connects its inoutput with another module's inoutput. Fig. 1 shows the connectors of a personal assistant (called "PersonalAgent"), which is described section 0 below. "-" represents inoutput, "+" – input and "#" – output connectors. An OMiSCID service also defines variables, with read or read-write access, representing its state.

This architecture allows us to connect modules, developed possibly by different programmers, and to use each module like a black box, knowing only the message format (see section 0 below). Moreover, this facilitates the ubiquitous deployment of the system. In fact, modules can run on different hosts, including mobile devices: OMiSCID and its modules can be implanted 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, applications run on one host, but are ubiquitous because of their easy and transparent access to any other mobile or stationary device of the environment, "equipped" with a module providing access to it.
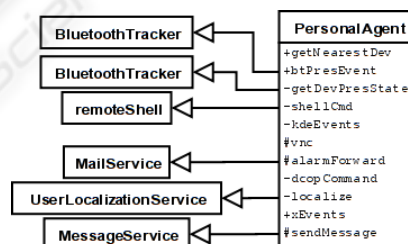
Fig. 1. The personal assistant and his connectors. Each connector is used for communication with a particular OMiSCID service. (Not all the connections are shown.)

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 (www.castor.org) 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.

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 maintain. For instance, a person tracker runs on several machines to monitor several rooms. We need an easy way to install and update the tracker on all these machines when a new version is developed.

Furthermore, we are in an environment with numerous stationary machines and where devices 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 provides these functionalities. We use Oscar as an implementation of OSGi. An Oscar platform runs on each device of the ubiquitous environment. At least two bundles are required: one incorporating jOMiSCID, a Java implementation of OMiSCID[1] and one incorporating Castor for modules to decode messages they send and receive (section 0). All the modules are at the same time OSGi bundles and OMiSCID services. A common bundle repository on a central machine regroups all the modules. It is accessed via http. Every Oscar platform installs its bundles from this repository and thus is linked to it. Newly developed modules are added to the repository and the Oscar platforms just install the corresponding bundle. When a module is updated in the repository, the Oscar platforms just update their bundle. This is convenient because once this architecture is set up, new modules are 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.

For Easy Development: A Wizard. To facilitate the programmer's work, we have developed an Eclipse plug-in 11: a wizard for creating new projects of type OMiSCID service and OSGi bundle. An empty project generated with this wizard contains

---

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

the correct arborescence of packages and files, the main classes with the OMiSCID code to create and register the service, the XML file for the developer to specify the connectors, the lifecycle code for OSGi, the manifest file and the build files to build and publish the new bundle on the repository. 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 OMiS-CID service is visible in the domain.

A user interface has been created to visualize the services running in a domain. It is shown Fig. 3. Fig. 2 shows the graphical interface of Oscar.

Fig. 2. The graphical interface of Oscar, showing the bundles. The active bundles are visible as services on Fig. 3.
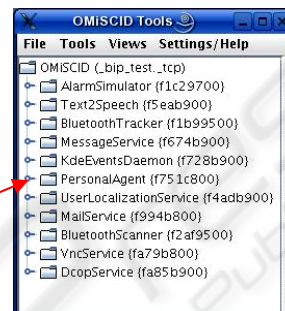
Fig. 3. Visualization of the OMiS-CID services running in a domain.

3.3 Example: A Personal Ubiquitous Assistant

We use the architecture described above (section 0) for a personal ubiquitous assistant. Our environment is equipped with a number of devices used to retrieve information about the user and his context. Mobile devices brought by users contribute to that knowledge. We use that knowledge to propose relevant services to the user. The assistant is the central module of the system. The assistant can do task migration: doing a task instead of the user. It can also provide additional services, such as forwarding a reminder to the user while he is 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 and these services are *user-adapted*. In fact, most of current work on pervasive computing 15, 12 fires pre-defined services in the correct situation. We start with a pre-defined set of actions and adapt it progressively to each 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.

The assistant observes the user and gathers clues on his context and his activity. For that he needs perceptive modules – sensors. For providing services to the user, it needs proactive modules – effectors. Some modules run on any host and some are

linked to, or used to access, specific hardware. They run on the machine connected to it. For instance, a person tracker needs at least one camera. The person tracker module is installed on the machine connected to the camera(s). Likewise, each machine equipped with speakers would have a speech synthesizing module. To sum up, our ubiquitous system is composed of several modules and the personal assistant. Sensor modules fire events received by the assistant, or the assistant explicitly interrogates a sensor. With this input the assistant estimates the user's situation (section 0). The learned behavior indicates to the assistant how to act in a certain situation. It contacts effectors to execute the appropriate action (*e.g.* provide the chosen service).

Mechanism of the Assistant. We mentioned in section 0 above an example of a service provided by the assistant: the user's agenda fires a reminder while he is away. The system detects this event and reacts by informing the user of the reminder. It tries to find the user in the building and to send him a message. The assistant takes into account the user's context (whether he is alone or not) and preferences (his preferred modality for receiving a message). In the following sections, we explain in detail how this example works.

The Context Model

A context is represented by a network of situations 6. 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. A situation represents a particular assignment of entities to roles completed by a set of relations between entities. Changes in the relations between entities or the binding of entities to roles indicate a change in the situation. A federation of observational processes detects situation changes.

In order to provide services we define rules associating actions to situations. An action is triggered when a situation is activated. This corresponds to a service being provided to the user. Fig. 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.
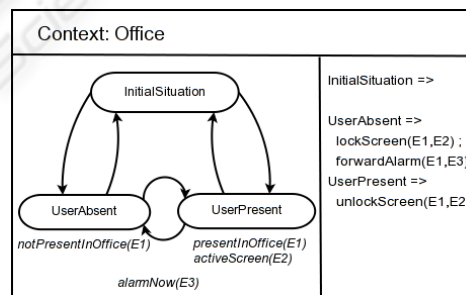
Fig. 4. Extract of the context model.

The Ubiquitous Database

All modules share a database divided into four parts: *user*, *service*, *history* and *infrastructure* (Fig. 5). Each part is implemented as an SQL schema.
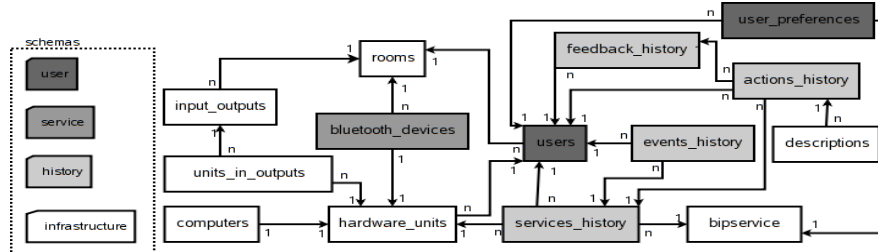

Fig. 5. Simplified scheme of the database.

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 system. The part *infrastructure* contains known, static information about the environment: rooms of the building and their equipment (in- and output devices and the hardware units controlling them). This knowledge allows to determine the most appropriate device for a service. The part *user* describes registered users (associating users with their Bluetooth devices and storing user logins for identification). As a future evolution, this data will be brought in by the user's PDA as a profile. When the user enters the environment, his profile is loaded into the system. This discharges the PDA of any heavy treatment necessary to exploit this personal data (learning algorithms, HMMs, etc.). The computation is taken over by an available device and the PDA, whose resources are very limited, stays available for its primary purpose. When the user leaves the environment, the new profile is migrated back on the PDA and the user has the updated data with him (to be used in another ubiquitous environment). The part *service* is a free ground for plug-in services. For instance, if available, Bluetooth USB adapters can detect users' presence thanks to their Bluetooth cell phones and PDAs. This information may be used to determine the identity of a video tracker's target.

The database is the knowledge and the memory of the assistant. For easy communication with the database, we use Hibernate (www.hibernate.org). We enclose it into a bundle used by other modules to query the database.

The Global Mechanism

Fig. 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 in order to reach the goal of providing services to the user. It is linked to the environment by the modules mentioned above (section 0). The latter run in the OSGi platform as bundles and communicate with each other (and with the assistant) 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 as the *role* (in the sense defined above, section 0) the user is playing. It sends it as an input to the context model by

asserting Jess facts. These possibly trigger situation changes. The activation of a situation triggers 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.
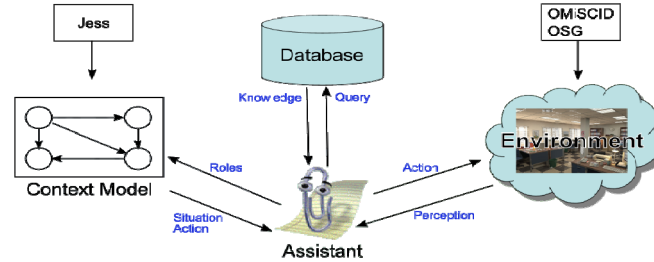

Fig. 6. The global mechanism of the system.

Let us reconsider the example scenario mentioned section 0 above in order to explain in depth how it works. The user's agenda (KOrganizer) triggers a reminder. A module ("KAlarmDaemon") listening to reminders detects this event. It sends a message on his output connector. The assistant 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. Otherwise, if the situation is "UserAbsent", 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. Thus one of these rules is fired 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. If the user is not to be found, the reminder is forwarded by email[2]. 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.), according to the user's context (alone or not). For each modality in preference order, the assistant queries the database for the machine it can be provided on, in the room the user is in. Then it searches on that host for the needed module and, if required, starts or even installs the corresponding OSGi bundle remotely. One can notice that the host used to forward the reminder could be a mobile device, for instance the user's PDA, as long as it is in a WiFi covered area.

4   Conclusion

We presented in this paper an architecture for ubiquitous applications. Our aim is to provide developers an easy way to build such systems. Looking at the requirements

---

[2] A text message on the user's mobile phone would be more adequate.

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.

References

1. Bisdikian, C., Christensen, J., Davis, J., Ebling, M. R., Hunt, G., Jerome, W., Lei, H., Maes, S. and Sow, D.. Enabling location-based applications. In *WMC '01: Proceedings of the 1st international workshop on Mobile commerce*, p. 38-42, 2001. ACM Press.
2. Coutaz, J., Rey, G.. Foundations for a Theory of Contextors. In *CADUI*, p. 13-34, 2002.
3. Dey, A. K. and Abowd, G. 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.
4. Emonet, R., Vaufreydaz, D., Reignier, P. and Letessier, J.. 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.
5. Hohl, F., Mehrmann, L. and Hamdan A.. *Trends in Network and Pervasive Computing - ARCS 2002: International Conference on Architecture of Computing System. 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.
6. Crowley J. L., Coutaz J., Rey G. and Reigner P.. Perceptual components for context awareness. In *International conference on ubiquitous computing*, p. 117-134, 2002.
7. Bardram J. E.. *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, p. 98-115, May 2005.
8. Lei, H., Sow, D. M., Davis, J., Banavar, G. and Ebling, M. R. The design and applications of a context service. *SIGMOBILE Mob. Comput. Commun. Rev.*, 6(4):45-55, 2002.
9. Letessier, J. and Vaufreydaz, D.. Draft spec: BIP/1.0 - A Basic Interconnection Protocol for Event Flow Services. www-prima.imag.fr/prima/pub/Publications/2005/LV05, 2005.
10. Pascoe, J.. Adding Generic Contextual Capabilities to Wearable Computers. *iswc*, 1998.
11. Reignier   P..   Eclipse   Plug-in   creating   an   OMiSCID   project.   www-prima.imag.fr/reignier/update-site.
12. Ricquebourg V., Menga D., Durand D., Marhic B., Delahoche L. and Log C. The Smart Home Concept: our immediate future. In *Proceedings of the First International Conference on E-Learning in Industrial Electronics*, Hammamet - Tunisia, December 2006.
13. Schilit, B. N., Theimer, M. M. and Welch, B. B. Customizing Mobile Application. In *USENIX Symposium on Mobile and Location-independent Computing*, p, 129-138, 1993.
14. Spreitzer, M. and Theimer, M.. Providing location information in a ubiquitous computing environment (panel session). In *SOSP '93: Proceedings of the fourteenth ACM symposium on Operating systems principles*, p. 270-283, New York, NY, USA, 1993. ACM Press.
15. 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.
16. Weiser, M.. The computer for the 21st century. *Scientific American*, p. 66-75, Sep. 1991.