Building Smart Appliance Integration Middleware on the OSGi Framework

Hiroo Ishikawa, Yuuki Ogata, Kazuto Adachi, and Tatsuo Nakajima

Department of Computer Science
Waseda University
{ishikawa,ogata,karvolta,tatsuo}@dcl.info.waseda.ac.jp

Abstract

The number of various kinds of everyday objects that contain embedded computers is increasing due to the popularity of ubiquitous computing. While component-based software development becomes common in a variety of application domains, ubiquitous computing requires component frameworks that offer more advanced features than the current component frameworks. This paper explores a component framework in the context of a ubiquitous computing system. We designed and implemented a system coordinating various home appliances, called SENCHA. SENCHA is built on the OSGi component framework. Although the framework provides facilities for dynamically changing environments, this paper reveals several problems of the framework in our experience.

1 Introduction

Various objects are empowered by embedded computers and network connectivity. It makes system software, such as operating systems and communication middleware, widely adopted from dusts[11] to a town[4]. On the other hand, multiple ways to use the embedded computers are required to satisfy various kinds of users' requirements or even dynamically changing requirements. Thus, software infrastructures for ubiquitous computing have to take into account the heterogeneity of computers, networks, environments, and users.

A lot of ubiquitous computing systems have adopted component-based software development, because the modularization of systems makes them portable and flexible. The system's portability is an effective characteristic in the heterogeneous platforms. As the ubiquitous computing systems do, software component technologies also need to consider the heterogeneity of ubiquitous computing, because it is being deployed on not only enterprise and desktop platforms, but also various embedded platforms[6]. The embedded computers have basically poor resources on CPU power, size, and so forth. Thus, software components that form ubiquitous computing systems are required to deal with the resource constraints and other embedded system's characteristics, unlike ones in the current resource-rich environments. Furthermore, a lot of features and services such as *augmented reality*[10] and *context-aware computing*, will be available in a future ubiquitous computing environment. It is necessary for the software component technology to provide rich programming support by component frameworks and tools, and take into account resource constraints on the embedded systems.

This paper describes our experience with component-based ubiquitous computing system development. We have built a smart appliance integration system on the OSGi Component framework, or the OSGi framework[14]. The system called SENCHA, integrates various kinds of smart appliances and services around the user. The integration is handled based on the user's preference and situation. Although the OSGi framework provides several facilities for dynamic configurations, our experience with building the system revealed the limitation of the current component framework. This paper also presents some requirements for future component frameworks in ubiquitous computing environments.

The rest of this paper is organized as follows. In Section 2, the design and implementation of SENCHA are presented. Section 4 discusses the experience with building SENCHA on the OSGi framework, and shows some requirements for future component frameworks in ubiquitous computing environments. In Section 5, related work is described in terms of software component technologies and their case studies in the context of ubiquitous computing. Finally, Section 6 concludes the paper with our future directions.

2 SENCHA: Coordination of Smart Appliances

Ubiquity of smart appliances and services makes users difficult to find and choose the services that will satisfy the user's requirements at every moment. In order to solve this problem, SENCHA addresses dynamic integration of the heterogeneous services and appliances according to every user's requirements and context. SENCHA focuses on the following three technologies: *service discovery*, *appliance information filtering*, and *appliance control*.

A system support is necessary for users to utilize the ubiquitous appliances in an ad hoc manner according to each user's requirement and situation. Unlike consumer electronics, it is difficult to find any other computer-enhanced everyday things such as cups, furniture, and walls accessible through networks. Moreover, even though a user knows that these things are accessible, it is also difficult to choose appropriate appliances or services provided by them to satisfy his current requirement, because there are a lot of smart appliances around him. Moreover, functionalities of the smart appliances are different, even if the appliances belong to the same category such as TVs, microwaves, and kiosk terminals. For example, a microwave provides users a dial interface to set the time to cook but another provides up and down buttons for doing the same thing.

2.1 Architecture Overview

Since the appliance coordinations are different by each user, we have adopted personal devices such as mobile phones and PDAs as the appliance coordinator. The benefits of this architecture are to be able to isolate personal information from environments and other users, and to utilize the personal information to customize the coordination. Appliance discovery, presentations, and controls are dealt with mainly by the coordination devices.

SENCHA provides three elements: *Personal Appliance Coordinator (Coordinator or PAC)*, *Display Service*, and *smart appliances* (Fig.1). Coordinator resides in a personal device, and moves from space to space with the user. It aggregates the appliance information, generates user interfaces, and controls the appliances.

Each of the smart appliances provides more than one services. They have own information (e.g. functional and non-functional specifications and state information). They are searched by Coordinators, but it can advertise their existence. When a user gets into an environment, his Coordinator collects each smart appliance's information by using a service discovery protocol.

Display Service is used for interactions with the user. When the user moves closer to Display Service, the Display Service detects the Coordinator by using sensors, then requests by the Coordinator appliance control interface in that situation, and then presents a user interface based on the user's preference, experience, situation, and so forth. The user can access to each of the smart appliances or a set of the smart appliances through the automatically generated user interface on the Display Service.

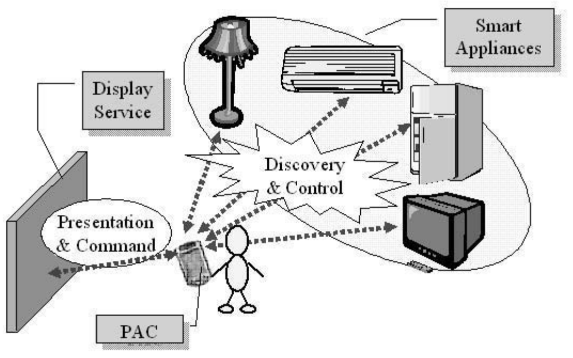

Figure 1. Architecture of SENCHA

3 Design and Implementation

3.1 Design Goals

SENCHA integrates various appliances and services around the user. There are a lot of environmental differences. Therefore, SENCHA takes into account the following three design goals:

Reconfigurability of systems: For the sake of reconfigurability, the system has to be divided into well-separated components. Coordinator is required to work in heterogeneous environments where different appliances, sensors, and protocols exist. The reconfigurability of the Coordinator is a crucial feature to integrate appliances at any environments. Furthermore, for time- and situation-awareness, the system is required to change its behavior dynamically.

Portability of components: Some functionalities such as device discovery are common among Coordinator, smart appliances, and Display Service. Portability or reusability of components is required for those components. Such components have to be independent of other components as much as possible.

Scalability of systems: There will be a lot of appliances in ubiquitous computing environments. Coordinator on a personal device has to deal with a large amount of appliances and services, and organize them.

3.2 Component Design

Personal Appliance Coordinator: Figure 2 shows the architecture of the Personal Appliance Coordinator (Coordinator) which organize heterogeneous smart appliances. Coordinator consists of five components: *Sonar* searches smart appliances and advertises the existence of Coordinator. *Appliance Registry* stores every smart appliance information that Coordinator has found. *UI Constructor* and *Context Inference Engine* generates integrated user interfaces based on the informations in the Appliance Registry and the user's preference and situation. *Appliance Controller* receives commands from Display Service, and dispatches the command to the specified appliances.

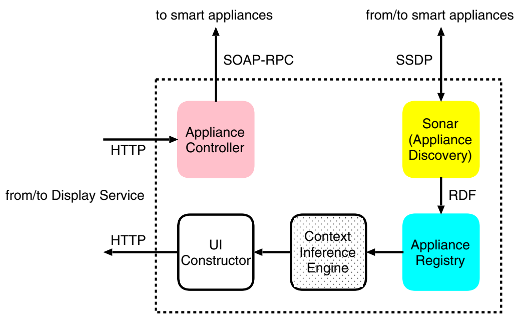

Figure 2. Architecture of Personal Appliance Coordinator

Each of five components is constructed of smaller components. For example, Sonar, which is an appliance discovery component, consists of a device discovery protocol stack, and a discovery message handling component.

The configurations of Coordinator can be changed by replacing components. This is one of the benefits of component-based software if components are well separated into clear roles. For example, the Context Inference Engine component can be replaced for changing strategies or methodologies of context inference according to the user's requirement, or a security policy for respective environments. These configurations can be dynamically applied by the OSGi framework.

Display Service: Display Service offers user interface for providing services integrated by Coordinator. Display Service consists of sensing and display components (Fig.3). In order to detect users near Display Service, the sensing component uses user location sensing devices such as motion sensors, infrared sensors, and RFID tag readers. The display component is triggered by the sensing component, and then requests appliance information to Coordinator detected by the sensing component. Operations on the display are delivered to the user's Coordinator by the display component. The operations are finally dispatched to appropriate appliances by Coordinator.

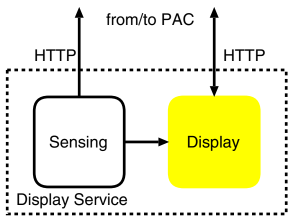

Figure 3. Architecture of Display Service

Smart Appliances: Each smart appliance consists of three components: Sonar, Control Point, and device drivers. Sonar is used for only service advertisement, unlike that of Coordinator, because smart appliances only provide services for users, and the appliance management is entirely handled by Coordinator. Control Point is an access point from Coordinator or any other smart appliances.

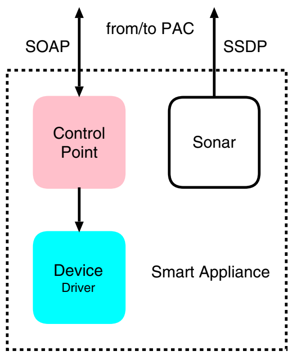

Figure 4. Architecture of smart appliances

3.3 Implementation

All three elements of SENCHA are implemented on the OSGi framework. The following sections introduce the OSGi framework, and describe the implementation of SENCHA in detail.

3.3.1 Overview of the OSGi Framework

The OSGi framework is a part of the OSGi Service Platform[14]. The framework supports dynamic component deployment, component dependency management, and component lifetime management. Software components in the OSGi are dynamically installed on the framework by specifying the location of the component in Uniform Resource Locator (URL) formats. The installed components are cached locally, thus even if the framework is terminated accidentally, it can restart the last installed components. The current OSGi framework deals with component dependencies as the relationship between components. The components can provide multiple services. The dependencies among services, however, are not supported by the framework. A component has one of six states on the framework during its life time, such as INSTALLED, RESOLVED, STARTING, ACTIVE, STOPPING, and UNINSTALLED. State transitions are triggered by the framework.

A software component on the OSGi framework is called *bundle*. A bundle is a unit of deployment. Each bundle is a single JAR file, and is implemented in Java. A bundle must includes a *manifest file* in which the bundle name, inter-component dependencies, and others are described. The manifest file is extended for the OSGi framework, for example the *bundle-activator* field. A bundle may provide zero, one, or multiple services. The services are defined as interfaces in the Java language, and registered to an internal bundle registry of the framework by their bundle activators. Each bundle activator is implemented by a bundle in order to perform customized operations when the bundle is started and stopped.

The implementation of the OSGi framework that we have adopted is Oscar[13]. Oscar is an open source implementation of the framework in pure Java. The developers of Oscar maintains the Oscar Bundle Repository on the Internet that contains various useful off-the-shelf bundles.

```
<!-- subject -->
<rdf:Description rdf:about="urn:sencha:TV1">
  <!-- predicates and objects -->
  <ssdp:serviceName>
    urn:sencha:TV1
  </ssdp:serviceName>
  <ssdp:serviceType>
    upnp:sencha:TV
  </ssdp:serviceType>
  <ssdp:location>
    :8080/axis/property.rdf
  </ssdp:location>
  <ssdp:cacheControl>300</ssdp:cacheControl>
  <core:friendlyName>TV</core:friendlyName>
  <core:room>roomA</core:room>
  <core:owner>all</core:room>

  <core:functionType
    rdf:resource=
      "urn:homecomp:function:TVFunction" />
  <core:function>TV</core:function>

  <core:URLBase></core:URLBase>
  <core:WEBPort>:8080</core:WEBPort>
  <soap:controlURL>
    /axis/services/LightService
  </soap:controlURL>
</rdf:Description>

<!-- subject -->
<rdf:Description
    rdf:about=
      "urn:homecomp:function:TVFunction#power">
  <!-- predicates and objects -->
  <function:value>on | off</function:value>
  <function:mean>turn power</function:mean>
  <function:method>power</function:method>
</rdf:Description>
```

Figure 5. An example of the appliance function specification

3.3.2 Device Discovery

Device discovery is performed between smart appliances and Coordinator, and Display Service and Coordinator. In any cases, the Sonar component is the only one component to deal with the device discovery message. Sonar component is installed in all three elements. The current implementation of Sonar adopts Simple Service Discovery Protocol (SSDP)[9]. The SSDP protocol stack is contained as a component in the Sonar component.

Appliance information is described as an RDF (Resource Description Framework) document[15]. The RDF document can be expressed by using the following three elements: *subject* (or *resource*), *predicate* (or *property*), and *object* (or *value*). The system can search appliance information not only by using appliance name, but also by using various kinds of properties. Therefore, this is very useful to describe a service and its functionalities or characteristics. Each appliance has own resource description in RDF-XML file. Figure 5 shows an example of the specification document for a TV appliance.

SENCHA uses the LOCATION field in the Alive and the Response messages in order to specify the logical locations of the appliance specifications. The location is described in URL. After receiving these messages, Coordinator downloads the specifications from each URL.

Appliance Registry is implemented as an RDF database developed by CodeRidge project[12]. The RDF database stores the appliance information in an *RDF triple* format. Therefore, the specification is transformed from RDF-XML format to RDF triple format in order to store it in Appliance Registry. An RDF triple is a statement that consists of a subject, a predicate, and a object. In our example shown in Figure 5, the RDF file has 18 triples.

3.3.3 Automatic User Interface Construction

Coordinator contains an HTTP server component in order to generate user interface automatically, and receive URL-based commands(described in 3.3.4) from Display Service. Since Oscar Bundle Repository provides HTTP Service, which is a set of components providing an HTTP server and a servlet engine, we reuse the components as the HTTP server components.

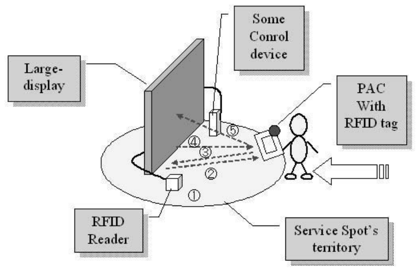

Figure 6. The procedure of user detection and UI construction

Since we have adopted URL-based commands to operate smart appliances, we can use various document formats that can embed URLs in a document, such as HTML, Flash and Microsoft Power Point and so on. The current implementation of UI Constructor supports HTML documents. The information used for the user interface are the name of appliance, command names and the attributes of the commands.

Context Inference Engine customizes user interface according to the user's preference and situation. Context Inference Engine is placed between the Appliance Registry component and the UI Constructor component, and filters the appliance information based on the preference and several sensor data. This component has an internal context database to store rules to infer the user's context. Currently, we are designing a new approach to solve the difficulty of constructing many rules. In our approach, we assign each customizing rule to an RFID tag. These tags are read by the RFID reader on the personal device. Then, rules are registered to Context Inference Engine. Therefore, the user can get free of writing his/her rules one by one. The rules are transformed to a filter, written in the Prolog, and a Prolog interpreter chooses appliance information by using the filter.

3.3.4 Appliance Control

The appliance control is performed among Display Service, Coordinator, and each smart appliance. Commands issued from Display Service are received by Coordinator. The Appliance Control component in Coordinator parses the commands, and generates SOAP stubs. Coordinator communicates with each of the appliances by using the SOAP stubs.

The appliance control commands from Display Service are encoded in URLs[8]. The standard URL syntax is extended for the control commands by a similar way to the syntax for CGI programs so that the path elements in a URL form contains some additional information for specifying and controlling smart appliances.

Each of the command request contains two elements. The first element is the device name that specifies the target appliance and it is also possible to use a query expression. A query expression is represented by a field-value pair and is beginning with the "`?`" attribute, for example, `?function=light` that indicates the appliance whose function is light. The second element specifies functions, and includes pairs of method name and its argument beginning with "`!`". The following line is an example of the URL-based command expression.

```
http://192.168.10.222/?
  ?function=light&type=floor/!power=on
```

This command requests for switching on lighting appliances on the floor. The command is transmitted to Coordinator that is assigned 192.168.10.222. The path elements following `/?` attribute represents a query and specifies the appliance whose function property is `light` and type property is `floor`. The `!power=on` element specifies `power` function with the `on` value. If the first element specifies a target appliance name, then the Appliance Controller component retrieves the appliance information from the Appliance Registry component. If the first element is a query beginning with `?` attribute, the Appliance Controller component requests the Appliance Registry component for the ones which have the corresponding property field-value pair within the query element.

SOAP is adopted communication between Coordinator and the smart appliances[16]. Each of the appliance information includes the location of a WSDL file which defines a SOAP stub[17]. Coordinator generates a SOAP stub from a WSDL file for each appliances.

The current implementation adopts only SOAP-RPC to access to the appliances, however it can communicate in other protocols such as CORBA and Java RMI in the future implementation.

4 Discussions

This section discusses our experience with building SENCHA on the OSGi framework, and proposes some requirements for future component frameworks.

4.1 Experience

Although the OSGi framework provides us several facilities such as dynamic component deployment, lifetime management, and so forth, it is still insufficient for the dynamically configurable systems. This section describes some lessons learned from our experience with the development of SENCHA.

Implicit Dependency: Component downloading facility is useful but causes the implicit dependency. The implicit dependency is defined as dependency that is related to component behavior, and not described in component interface or other descriptions such as the manifest files. In our case, the javax.xml package implicitly depends on the runtime environment. When Sonar runs on the framework on Java runtime version 1.4, there is no problem because the Java runtime 1.4 natively provides the javax.xml package. However, when Sonar runs on Java runtime version 1.3, since it doesn't provide javax.xml package, So, we had make a bundle that contains javax.xml package. However, since the OSGi framework assigns a class loader to each bundle instance, and the javax.xml package assumes that it is instantiated in the same space as the applications, the javax.xml bundle didn't work properly. We solve this problem by installing the javax.xml package to the external library directory of the Java runtime. Consequently, there is implicit dependency between the implementation of the bundles and the bundle loading mechanism.

Resource Constraints: Although the OSGi framework manages the lifetime of components, it doesn't take into account resources that components consume. Three elements of SENCHA (Coordinator, Display Service, and smart appliances) will run on embedded systems. Thus, the resource constraints significantly affects system configuration strategies.

However, this is also a problem of the Java Virtual Machine. The components on the OSGi framework are run on JVM. Therefore, in order to manage the resources, it is necessary for the OSGi framework to import the resource management facilities (e.g. Real-time Java API) from the JVM.

Multi-Level Views: Multiple views of abstraction levels are necessary to be supported by the component framework. A component may consist of some components. For example, Sonar, which is an appliance discovery component, includes some other components such as the SSDP protocol stack and message handling components, and each of them is compressed in a JAR file. The framework recognizes a JAR file (bundle) as a minimum deployment unit, thus we cannot dynamically configure the Sonar component by replacing the component inside it.

4.2 Requirements

From our experience with developing SENCHA on the OSGi framework, we present several requirements for future component frameworks in a ubiquitous computing environment. Although each requirement might have been already claimed by others such as Architecture Description Languages (ADLs)[7], we have found that these requirements in the context of the ubiquitous computing systems are very important. Therefore, these requirements must consider as the essentials of the future component frameworks.

Connector Abstractions and Their Configurations: Connector abstraction, or simply, connector is a component as glue between other components. The connectors are adopted by many ADLs. Although ADLs claim that the benefit of the connectors make the relationship among components explicitly, we notice that the connector is useful for changing relations between components. The connectors facilitate to change the relations between components. For example, we can change from a local connection between components to the TCP/IP connection by changing the connector from a local connector to a TCP/IP connector without generating complicated stubs and proxies.

Moreover, the connectors are useful for calculate memory consumption during configuration time. The memory usage is very important information for embedded systems because some of them have to operate applications without a secondary storage for saving data. Since the connectors make the relationship among components explicit, the framework can know components currently activated and cooperated.

Dependency Management: In order to realize heterogeneous configurations, inter-component dependency must be taken into account by a component framework for ubiquitous computing. So, dependency management is a crucial feature for such dynamically changing environments.

There are two types of dependencies between components: explicit dependency and implicit dependency. While the explicit dependencies are described in every component's interface, manifest file, etc, the implicit dependencies are related to the implementation or behavior of components[5]. The implicit dependency is occurred by the difference between an expected characteristic and the real characteristic of a component. For example, as described above, it becomes an implicit dependency that whether a bundle is instantiated on the OSGi framework or under the framework. Another example mentioned by [5] shows that thread scheduling policies and network reliability can be the implicit dependencies among components. The purpose of the dependency management should make the implicit dependency explicit.

5 Related Work

Connector abstraction is discussed in Architecture Description Language (ADL) researches[7]. ArchJava, which is one of the ADLs, provides the connector abstraction as a language construct. The case study conducted by the authors have built an ubiquitous computing system in ArchJava with the connector abstractions[1]. It is pointed out by the paper that program understanding, the correctness of implementation, and software evolution are improved by connectors. Although the paper mentioned the configurability in terms of introducing software evolution, they didn't mention the usage of the connectors in the dynamic nature of ubiquitous computing, and didn't describe the dynamic configuration of a system. ArchJava works as a preprocessor of a Java compiler, thus runtime functionality is limited by the Java virtual machine. In the context of ubiquitous computing, a component framework is required to support dynamism of systems, for example, dynamic component deployment by the OSGi framework.

Beanome is a thin layer constructed on the top of the OSGi framework and is motivated by the limitations of the OSGi framework[2][3]. The authors of the paper also point out the necessity of dependency management on the framework. They classify the limitations in the following three points: bundle-to-package, bundle-to-service, and service-to-service dependencies. According to the papers, bundle-to-service and service-to-service dependencies are implicit dependencies in the current OSGi framework. The implicit dependency that we have pointed out is classified into the service-to-service dependency. In order to resolve the implicit dependencies, the authors propose Beanome. Beanome adds component types to each components. The component types are defined in extended OSGi manifest files described as XML documents. The Beanome component type description allows us to describe required services in detail by using the regular expression, but it doesn't care the order of activation as this paper mentioned. Also, there is no field for describing resource consumption in the component type description.

6 Conclusion and Future Directions

This paper has explored the development of an ubiquitous computing system on the OSGi framework, and has shown the experience and some requirements for future component frameworks. The systems for ubiquitous computing is necessary to be dynamically configurable and extensible due to the heterogeneity and the mobility of smart appliances and users. The OSGi framework allows us to deploy components dynamically, and consequently configure a system dynamically. We have designed and implemented SENCHA that consists of three elements: Personal Appliance Coordinator, Display Service, and Smart Appliances. Each of the elements consists of components, and runs on the OSGi framework. As a result of our experience with building SENCHA, ubiquitous computing system requires a component framework that provides advanced features such as connector abstraction and dependency management. We are designing a new component framework satisfying the requirements presented in this paper. Our component framework aims at supporting: *connectors*, *component behavior description* that makes the implicit behavior explicit, and *dynamic configuration* including automatic component selection and deployment according to the inter-component dependency, the component behavior, and the characteristics of platforms.

References

[1] Jonathan Aldrich, Vibha Sazawl, Craig Chambers, and David Notkin. Language Support for Connector Abstractions. In proceedings of the European Conference on Object-Oriented Programming, July 2003.

[2] Humberto Cervantes and Jean-Marie Favre. Comparing JavaBeans and OSGi Towards and Integration of Two Complementary Component Models. In proceedings of the 28th Euromicro Conference on Component Based Software Engineering, September 2002.

[3] Humberto Cervantes and Richard S. Hall. Beanome: A Component Model for the OSGi Framework. In proceedings of the Software Infrastructure for Component-Based Applications on Consumer Devices, September 2002.

[4] Cooltown research. `http://cooltown.hp.com/research/`

[5] Hiroo Ishikawa and Tatsuo Nakajima. A Case Study on a Component-based System and its Configuration. In proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems, September 2003.

[6] Marija Mikic-Rakic and Nenad Medvidovic. Adaptable Architectural Middleware for Programming-in-the-Small-and-Many. In ACM/IFIP/USENIX International Middleware Conference, June 2003.

[7] Nenad Medvidovic and Richard N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. In IEEE Transactions on Software Engineering, vol.26, No.1, January 2000.

[8] Tatsuo Nakajima and Ichiro Satoh. Personal Home Server: Enabling Personalized and Seamless Home Computing Environment. In proceedings of Percom 2004, 2004.

[9] Yaron Y. Goland, Ting Cai, Paul Leach, and Ye Gu. Simple Service Discovery Protocol/1.0, IETF Internet Draft, October 1999.

[10] Eiji Tokunaga, Andrej van der Zee, Makoto Kurahashi, Masahiro Nemoto, and Tatsuo Nakajima. Object-Oriented Middleware Infrastructure for Distributed Augmented Reality. In proceedings of the 6th IEEE International Symposium on Object-oriented Real-time Distributed Computing, May 2003.

[11] B. Warneke, M. Last, B. Leibowitz, K. S. J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. IEEE Computer Magazine, vol.34, no.1, January 2001.

[12] CodeRidge. `http://sourceforge.net/projects/coderidge/`

[13] Oscar: Open Service Container Architecture. `http://oscar-osgi.sourceforge.net`

[14] OSGi Alliance. OSGi Service Platform Release 3, IOS Press, December 2003.

[15] Resource Description Framework. `http://www.w3.org/RDF/`

[16] Simple Object Access Protocol. `http://www.w3.org/TR/SOAP/`

[17] Web Service Description Language. `http://www.w3.org/TR/wsdl/`