Context-Aware Service Composition for Mobile Network Environments*

Choonhwa Lee[1], Sunghoon Ko[1], Seungjae Lee[1], Wonjun Lee[2,**], and Sumi Helal[3]

[1] College of Information and Communications,
Hanyang University, Seoul, Republic of Korea
{lee,archit01,mixxer}@hanyang.ac.kr
[2] Dept. of Computer Science and Engineering,
Korea University, Seoul, Republic of Korea
wlee@korea.ac.kr
[3] Dept. of Computer and Information Science and Engineering
University of Florida, FL, U.S.A.
helal@cise.ufl.edu

Abstract. Recent advances in wireless and mobile networking technology pose a new set of requirements and challenges that are not previously thought of, when it comes to smart space middleware design. Leading the list is how to embrace diversity and unpredictability inherent in mobile computing environments. Service-oriented computing is being recognized as one of viable solutions to the problem. According to the paradigm, dynamic service discovery and composition should be able to handle the dynamism and diversity in the environments. However, most current service frameworks do not provide sufficient support to mask the complexity from having to deal with the uncertainty by ourselves. Therefore, building an application via qualified service composition still remains a cumbersome and daunting task. In this paper, we present a smart space middleware architecture designed to hide the complexity involved with context-aware, automated service composition. We also report our prototype implementations as an effort to validate the effectiveness and feasibility of the architecture.

1 Introduction

Service-Oriented Computing (SOC) has widely been recognized as a viable solution to cope with the dynamics inherent in mobile and pervasive computing environments. Embracing unpredictable changes requires that the computing system be open and extensible to ensure its evolution, adaptive to environmental changes, and flexible enough to allow its reconfiguration. It has been shown that these crucial properties of the ubiquitous computing middleware can be enabled, in essence, by the concept of services [1] [2] [3] [4]. According to the service paradigm, everything is modeled as a

* This work was supported by grant No. R01-2005-000-10267-0 from the Basic Research Program of the Korea Science & Engineering Foundation.
** Corresponding author

service, including hardware devices, network resources, a piece of computation, and even a human being. A value-added, composite service can also be formed by interconnecting those component services that provide limited functionality. Being based on the service concept implies dynamic service discovery and late binding whereby desired application functionality is mapped to appropriate module(s) available in the environment at the moment. Moreover, this mapping may be replaced with what is considered a more desired one later on. This concept of services renders the architecture effective in dealing with the future unpredictability.

Originally introduced as a home gateway platform, OSGi Service Platform [5] provides a managed service execution environment on which services can dynamically be installed, invoked, and then uninstalled. Being similar, in spirit, to the microkernel approach, it allows for extendable, flexible system configuration, to which OSGi's wide acceptance might be attributed. Although OSGi specification underwent rapid growth over the past few years, some advanced features related to service use and management remain yet to be further developed to date. Especially, we notice that service composition is one of such desired features.

This paper presents our work on an OSGi-based smart space middleware architecture with its emphasis on context-aware service composition. We first discuss our design principles and overall architecture of the smart space middleware, before delving into the main issues of service discovery and composition assisted by context awareness.

2 Architectural Design of Smart Space Middleware

Recent advances in wireless and mobile networking technologies for small-sized networks [6] [7] pose a new set of requirements and challenges that are not previously thought of, when it comes to smart space middleware design. The examples of small networks we consider include PAN (Personal Area Network), VAN (Vehicle Area Network), and home networks. Their size constraint is inevitably passed on the middleware that likely resides on a gateway node in the networks. Therefore, the requirement of being lightweight middleware is one of the most critical. At the same time, the middleware architecture should be flexible and extensible enough to accommodate a range of applications over various network configurations. While designing the middleware architecture, its configuration flexibility was regarded as a key feature to facilitate interactions and collaboration among middleware components. Our smart space middleware has been designed on top of OSGi Service Platform to address those primary requirements. It is worthwhile to note that the OSGi Alliance has also established Mobile Expert Group and Vehicle Expert Group to explore the possibility of OSGi for mobile devices and in-vehicle environments [8].

This paper focuses on the issue of context-aware, automated service composition in a mobile network environment. By introducing an OSGi system service, named as Plumber Service, to support automatic service composition, a composite service can be embodied by a combination of best qualified instances available at the moment. After then, during service runtime, any broken and deteriorated composition can automatically be recovered and replaced, respectively. Context awareness (i.e., ser-

vice quality) enabled by our proposal of dynamic service ranking helps Plumber Service in providing this failure resilient service composition.

Figure 1 illustrates the overall architecture of our mobile space middleware. At the bottom are OSGi framework and system services. Our additions of network access services (Space Gateway, WiBro, WPAN, etc) and service composition services (Plumber) are positioned on top of them. All communication need is served by Space Gateway service that passes communication requests on to appropriate network modules such as WiBro service, WPAN service, or others. When building up a composite service as described by a service composition graph, Plumber Service figures out what would be the best combination by the help of the dynamic service ranking. At the top are shown applications such as Service Browser & Composer, Space Monitor, Preference & Capability Policy Manager, healthcare application, and so on.

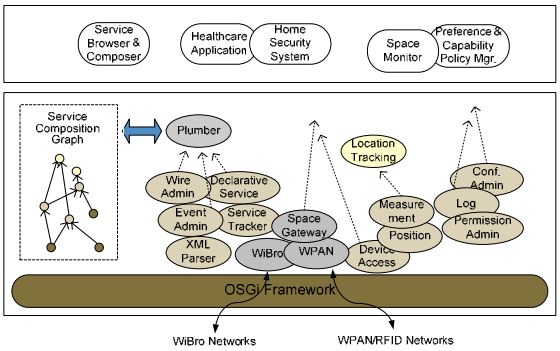


Fig. 1. Overall Architecture of Smart Space Middleware

3 Automated Service Composition Support in OSGi Environments

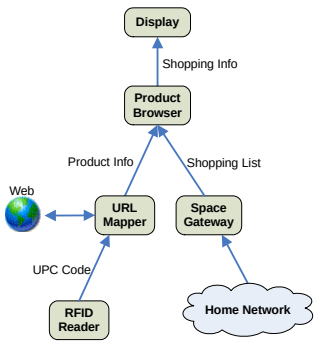
The latest OSGi specification [5] offers some advanced features related to service monitoring and composition: Wire Admin Service, Service Tracker, and Declarative Service. First, Wire Admin Service defines a set of APIs that can be used to wire a pair of services, providing support for the producer-consumer design pattern. Service Tracker offers assistance for service availability tracking beyond the low-level event mechanism of OSGi Service Platform that notifies which bundle has become available, unavailable, or modified. Declarative Service is a new addition to OSGi specification release 4.0 that helps to manage service dependency [5] [9]. Basically, it provides system support for automatic service dependency management. According to the scheme, service dependencies are described in a declarative way, i.e., separate from service logic itself. In other words, by filling in the callbacks of a component's bind and unbind, developers can program actions that are taken to automatically handle changes to service availability the module depends on.

Despite those handy features for service availability checking and composition by the current OSGi specification, one missing piece we see is system support for automated service composition. As we will see below, a composite service is described as an interconnection of component services. Plumbing the components is a complex

task, involving several issues such as service discovery, selection based on availability and quality, and continuous monitoring. For example, any component turning out unavailable in the middle of composition process will require that the half composed services be unwired. Imagine, also, what to do if one link of initially optimal composition is clogged up later. In this paper, we propose Plumber Service that takes care of much of the intricacy involved in service composition and maintenance.

3.1 Shopping Aid Scenario: A Case of Service Composition

Before getting into the details of our Plumber Service, let us first consider an exemplary scenario of shopping aid application. The application is constructed by assembling several component modules as shown in Figure 2 (a). The graph shows information processing and flows over an interconnection graph of component services that provides basic functionality. Each node in the graph represents a component module, and information being passed to its successor node is shown over an edge between the two. The information is abstracted and aggregated, as it propagates over the graph. Suppose that Bob stops by a grocery store on his way home after work. As he walks through the store isles, shopping guide messages pop up on his PDA. This is made possible by a service composition depicted in the figure. First, his PDA equipped with a RFID reader reads in the UPC codes of grocery items on the shelves that he passes by. The codes are then translated into product information retrieved by URLMapper from manufacturer's Web sites. We assume that his PDA also retrieved a shopping list via SpaceGateway from the home fridge the moment he entered the store. ProductBrowser compares the product information and shopping list, and alerts Bob on the matches between the two by providing shopping information through Display.



(a) Service composition graph

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="shopping-aid" adaptation="yes" ... >
    <!-- adaptation attribute can be either yes or no. -->
    <comment>A shopping aid scenario</comment>
    <pipe name="p1" replacement="better">
    <!-- replacement attribute can be one of fixed, any, and better. --
>
        <producer>
                <name>org.kosef.service.ProductBrowser</name>
                <properties> language="en" </properties>
        </producer>
        <consumer> org.kosef.service.Display
                <properties>
                        language="en"
                        location=Harry's
                </properties>
        </consumer>
    </pipe>
    <pipe name="p2" replacement="fixed" sharing="permitted">
        <producer>
                <name>URLMapper</name>
                .....
        </producer>
        <consumer>org.kosef.service.ProductBrower</consumer>
    </pipe>
```

(b) Corresponding XML description

Fig. 2. Shopping Aid Scenario

To foster service composition, we have developed an XML-based service composition graph editor, called Service Composer, along with our Plumber Service. It is a GUI tool to guide us to walk through the service composition process from a graph specification to its plumbing to runtime management. More specifically, it helps to describe a composition by providing visual information about component services and a library of composite services currently available in the framework. It also performs compatibility checking of the producer and consumer service on both ends of a link on our behalf. Once the composer generates the graph in XML, actual instantiation of it is taken care of by Plumber Service. Without these supports, we will end up having to hard-code the graph construction by adding links leg by leg. Although the current OSGi specification provides some help with service composition as mentioned earlier, more can be done to relieve the burden of the composition complexity and unpredictability in mobile computing environments. It is important to know that the composed service is put back into the framework service registry, so that others can see and make use of the composite. Simplified part of the XML description corresponding to the graph in Figure 2 (a) is shown in Figure 2 (b). The important fields of the graph are explained below.

3.2 Plumber Service

Given a service graph specification, Plumber Service first makes sure that all necessary component services are readily available and there is no dependency problem by utilizing OSGi's support for service tracking and events. It, then, makes use of OSGi Wire Admin Service to interconnect the graph nodes, which implies that each graph edge is represented by an OSGi Wire object. Figure 3 shows Plumber interface that the Plumber Service implements. The interface defines *compose*() and *decompose*() methods for the purpose of building up and dismantling a graph. If a client wants to be notified of any change to the wiring, it should register callback methods for those events by calling *addPlumberListener*() method. It means that the client must implement PlumberListener interface as well.

```
public interface org.kosef.service.Plumber {
    String compose(String xml-graph-description);
    void decompose(String graph-name);
    void addPlumberListener(String graph-name, PlumberListener appl);
}

public interface org.kosef.service.PlumberListener {
    // Callback functions to be when the wiring has been
    // completed, a failure occurs, and the graph has been
    // recovered or part of it has been replace.
    String constructed();
    String broken();
    String reconstructed();
}
```

Fig. 3. Class Definition of Plumber Service

On system startup, Plumber Service registers itself with OSGi service registry to join the framework. After getting the handle of Plumber Service from the framework registry, a client calls *compose*() method on it by providing a service graph. The cli-

ent is notified through *constructed*() callback method in PlumberListener interface, when the graph instantiation is completed.

As shown in Figure 2 (b), *composite* element has *adaptation* attribute of which value can be either *yes* or *no*. It indicates whether a composite service should try to adapt itself to environmental changes during service runtime. The latter means that, once constructed, a component replacement with an equivalent is not permitted in the event of failures. Consequently, the breakdown of any part of the graph results in the composite service being flagged as down and unavailable. The former case mandates that the system service performs automatic recovery of the broken service graph. Problematic components should be replaced with an equivalent one or a substitute with better quality. (Service suitability can effectively be dealt with by the concept of our dynamic service ranking whose details are discussed below). An alternative can be substituted for a current yet still working participant that was the best choice at the time of service plumbing. In either case, the client application is informed of the changes in the graph via *broken*() and *reconstructed*() callbacks.

The *replacement* attribute of a *pipe* element specifies its replacement policy at the grain of an individual link: *fixed*, *any*, or *better*. The case of *fixed* says that the pipe must not be replaced even in case of the component failure, which leaves no choice but to dismantle the entire graph. The value of *any* means that the problematic link can be switched over to any other alternative. The *better* case allows only better replacements in terms of service quality. By the way, this *replacement* attribute precedes the graph-wise *adaptation* policy, in case there's a conflict between them.

3.3 Dynamic Service Ranking

Dynamic service ranking is our approach to the service selection problem that is about choosing the best when there are more than one qualified candidates. It is common that service property is indicated by service attributes attached to it. One can think of two types of static and dynamic attributes. For instance, a stationary service's location information is a static attribute which remains unchanged during the service lifetime, whereas frequently changing load on the service can be viewed as a dynamic attribute. We propose that contextual information be attached as an attribute to service instances, and the contextual property be dynamically updated to keep up with changes in the environments. Dynamically changing context information often plays a crucial role in avoiding unnecessary distraction in ubiquitous computing environments. Think about the distance between a moving person and display devices in surroundings. Perhaps, the best display may be determined by a combination of dynamic attributes (e.g., the display's location and orientation) and static attributes like the display size. In other words, we can narrow candidate display devices down to a few more qualified by evaluating the dynamic attributes at the time of service selection.

Like other service discovery frameworks, OSGi supports static attributes only. It defines SERVICE_RANKING as one of service registration properties that indicates service quality. The higher ranking value of the attribute, the better service quality it can provide. Therefore, a service with the highest ranking tends to be returned as OSGi's service lookup results, in case of more than one matches. While it should prove useful to some extent, it will not be capable of capturing the endless changes of

mobile computing environments. This is because it remains the same throughout service lifetime, once the value of static attributes is initially set by service providers. Consequently, an instance recommended by the service ranking may not be the best any more. This limitation can be overcome by our idea of dynamic service ranking that represents dynamically changing contextual aspects. The ranking property is dynamically evaluated either on demand or periodically. Code snippet in Figure 4 shows dynamic service ranking extension to OSGi specification. A service will provide its own implementation of DynamicRanker interface, if dynamic service ranking is supported by that. A Java thread may be dedicated to periodically re-evaluate the service quality every *refreshInterval* interval specified as an argument.

```
public static final String DYNAMIC_RANKING = "service.dynamicRanking";

public interface org.kosef.service.DynamicRanker {
    Integer evaluate(int refreshInterval); // returns service quality index
}

public class DynamicService implements DynamicRanker, BundleActivator {
    public void start(BundleContext context) {
        Hashtable properties = new Hashtable();
        properties.put(SERVICE_DESCRIPTION, "sample service");
        properties.put(DYNAMIC_RANKING, evaluate(0));

        ...

        context.registerService(this.class.getName(), this, properties);
    }

    Integer evaluate(int refreshInterval) {
        // It dynamically determines its rank by considering
        // any contextual information relevant to the service.

        return dranking;
    }

    public void stop(BundleContext context) {
        // stop the context evaluation thread
    }
}
```

Fig. 4. Dynamic Service Ranking Example

With the help of dynamic service ranking attributes, failure resilient service composition is now made possible. In the event of a fault or significant service quality degradation, the problematic component can be replaced with a better equivalent. It is on the value of the dynamic ranking attributes that our Plumber Service bases its recovery decision.

3.4 Service Composition Support Subsystem

Putting together those ingredients we discussed to this point, we have come up with an architecture of service composition subsystem. The architecture is illustrated in Figure 5, and it has been designed to make use of, whenever and wherever possible, existing OSGi features in support of service composition and management. The subsystem provides a visual service composition editing and management environment.

Service Composer on the top is an application program that produces a service composition graph specified in XML. The graph description is then passed to Plumber Service which orchestrates all the interactions for service composition and maintenance. Particularly, Plumber Service uses Wire Admin APIs to perform actual service wiring as directed in the graph. Once the wiring has been done, the availability and quality changes of participating component services are monitored through Event Admin service. Either periodically or on demand, whichever way is appropriate for the service in question, Dynamic Ranking Evaluator evaluates its dynamic ranking attribute to determine the value of the attribute. If the evaluation results in a new value, it is communicated to Plumber Service via Event Admin service. Also, the service's property in OSGi framework registry is accordingly updated. Being notified of any topological changes through Event Admin, Plumber Service may take recovery actions by issuing necessary wiring commands to Wire Admin service.

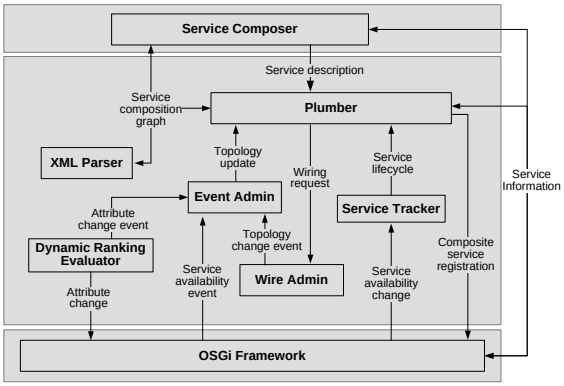
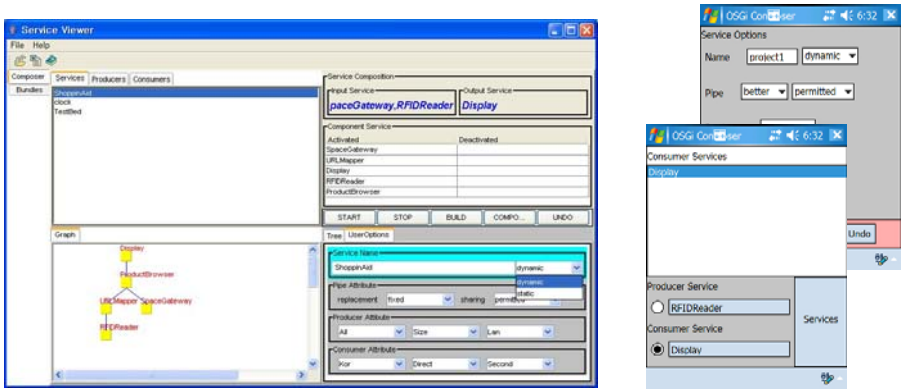


Fig. 5. Service Composition Subsystem Architecture

4 Prototype Implementation

To validate the effectiveness and feasibility of our smart space middleware architecture, we have developed two versions of its prototype for PC and PDA platforms, respectively. The former corresponds to the middleware being hosted on a gateway node in home/office environments, while the latter on a personal gateway node in a PAN/BAN. Both implementations are based on Knopflerfish v2.0 on top of JVM (J2SE 1.4.2 in the case of PC and IBM J9 CDC + Personal Profile v1.0 for PDA). Figure 6 shows some screenshots of them, while composing the shopping aid application depicted in Figure 2. The menu provides functionalities to retrieve and monitor system status with regard to individual services and their compositions and to change the configuration. Using the GUI, users can list up available services, including either of component and composite services, compose a new service, or change existing service attributes and compositions. It also provides the service management facility of service start and termination.



(a) PC version (b) PDA version

Fig. 6. Screenshots of Service Composer

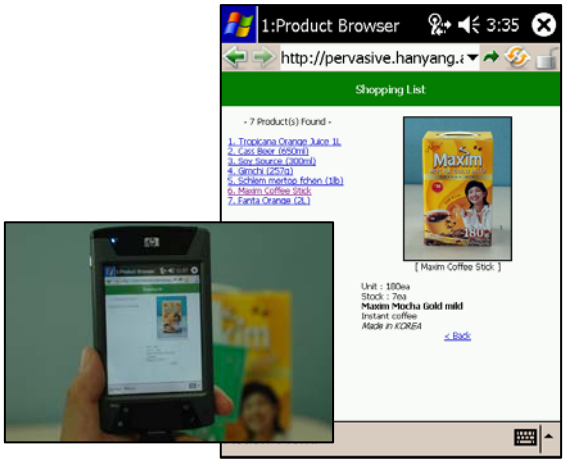


Fig. 7. Shopping Aid Scenario Demonstration

Figure 7 shows the shopping aid application running on Bob's PDA. As we can see in the figure, the purchasing list is being displayed on the left. As Bob passes by a product of the list in a grocery store, shopping information is cropping up on the right-hand side. He will be able to make a decision on whether to buy it or not at a glance. According to the service composition graph presented in Figure 2, Product Browser is linked to Display with a dynamic ranking attribute attached to it. The attribute evaluation ends up with a quality index indicative of how good the service is. The index value may be determined based largely on the distance between Bob and display devices. Suppose that Display service is embodied by PDA's display on entering the store. As Bob moves around, the mapping can be changed to a bigger display mounted on a wall or shelf in the store. In other words, Bob's movement affects the dynamic rankings of candidate display devices in the store, which leads to

Plumber Service replacing a failed or inferior component. The composition graph in Figure 2 includes all necessary information to instruct Plumber Service what to do, including whether to allow this adaptation based on observed environmental changes through the dynamic ranking.

5. Related Work

Service-Oriented Architecture (SOA) benefits us by allowing flexible and adaptive system configurations befitting a certain environment. Similarly to the micro-kernel approach, component services are discovered and plugged into the base composition framework when required. The plugged-in services can be woven into a composite service capable of providing a new, value-added functionality on the individual component's own. This composition problem has been explored under different context [3] [4] [10] [11] [12] [13]. Among recent salient efforts, in particular, in pervasive computing environments, are the exploitation of semantic service descriptions [14] [15] [16] and opportunistic combination/chaining of component services [17] [18] [19] beyond the previous, naive I/O matching scheme. Other research on the composition programming model and methodology [15] [20] is also worthy of notice.

Our work differs in that it addresses the need of service composition support especially for OSGi-based SOA framework. More specifically, an emphasis is placed on automated service composition, adaptation, and failure recovery made possible by being aware of relevant context to the composition. It is the dynamic service ranking attribute that expresses dynamically-changing contextual information in our case, while other noteworthy ones are also found. In fact, the concept of the dynamic ranking is similar to the idea of embedding context capturing modules into a service discovery framework [21]. In spirit, it may also be viewed as close to the dynamic URL [22] and dynamic attribute [13]. With the help of the dynamic service ranking attribute, our Plumber Service can not only provide an optimal composition solution under a given condition by taking into account relevant contextual properties. Once successfully built, it also takes care of the maintenance need of the composite such as partial replacements or failure recovery to adapt to changes to the environment. In sum, by providing system support for automatic service composition and maintenance, it eases the development process and operation of autonomous, complicated applications in pervasive computing environments.

Current OSGi specification [5] offers some level of supports for service availability monitoring and composition. They include Service Tracker (to be notified of changes to service availability), Declarative Service (to manage dependencies among services), and Wire Admin Service (to wire a producer-consumer pair together.) Although Service Tracker already provides basic assistance for tracking service availability, it has been shown that such support can be furthered in [23]. OSGi's Declarative Service has its root in Gravity project [9] where the problem of service dependency management is overcome by separating service logic itself from service management logic handled by the execution environment. A component description, including its dependency on other services it requires, is defined in XML documents (i.e., in a declarative way.) The execution environment manages bindings between the component and required services. Whenever it sees changes to its dependent service availability, the execution environment invokes bind and unbind callback methods (which is part

of the component description), so that the application can act on the changes. The project also addresses the need of service composition management support, which makes it similar to our work from the perspective of the level of service composition and automatic dependency management. Unlike Gravity project, however, our approach enables better adaptation to service availability by supporting mechanisms both for proactive actions (through the dynamic service ranking attribute) and for reactive responses as with in Gravity project.

6 Conclusion

This paper presents a service framework, especially designed for small-sized mobile networks, that can automate context-aware service composition. The two major contributions of our work are summarized as an architectural middleware design and its evaluation via prototype implementations. At the heart of the middleware architecture is Plumber Service that enables the best qualified service composition. After then, it keeps the composition optimal with the help of the dynamic ranking attributes that reflect the quality of the component service. The architecture has been designed to make use of, wherever possible, OSGi features to support service composition and management. Through the prototypes, we have demonstrated that the architecture achieves its main design goals of lightweight, flexible, failure resilient service composition middleware.

References

1. F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, “Unraveling the Web Services web: An Introduction to SOAP, WSDL, and UDDI,” IEEE Internet Computing, vol.6, no.2, , pp. 86-93, March-April 2002.
2. Sun Microsystems, Jini Network Technology, http://www.sun.com/jini.
3. S. D. Gribble, M. Welsh, R. von Behren, E. A. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Joseph, R. H. Katz, Z. M. Mao, S. Ross, and B. Zhao, “The Ninja Architecture for Robust Internet-scale Systems and Services,” Computer Networks, vol.35, no.4, pp.473-497, March 2001.
4. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, and K. Vahi., “Mapping Abstract Complex Workflows onto Grid Environments,” Journal of Grid Computing, vol.1, no.1, pp.25-39, 2003.
5. The OSGi Alliances, OSGi Service Platform Release 4, July 2006.
6. IEEE 802.15 Working Group, IEEE 802.15 WPAN, http://www.ieee802.org/15.
7. V. Devarapalli, R. Wakikawa, A. Petrescu, and P. Thubert, "Network Mobility (NEMO) Basic Support Protocol", RFC 3963, January 2005.
8. T. Giuli, D. Watson, and K. V. Prasad, “The Last Inch at 70 Miles Per Hour,” IEEE Pervasive Computing, vol.5, no.4, October-December 2006.
9. H. Cervantes and R. S. Hall, “Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model,” In Proc. of the 26th International Conference on Software Engineering (ICSE’04), pp.614-623, May 2004.

10. X. Fu, W. Shi, A. Akkerman, and V. Karamcheti, “CANS: Composable, Adaptive Network Services Infrastructure,” In Proceedings of USENIX Symposium on Internet Technologies and Systems (USITS), March 2001.
11. T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana, “Business Process Execution Language for Web Services Version 1.1,” http://www-128.ibm.com/developerworks/library/specification/ws-bpel/, May 2003.
12. A. Dey, M. Futakawa, D. Salber, and G. Abowd, "The Conference Assistant: Combining Context-Awareness with Wearable Computing," In Proceedings of the 3rd International Symposium on Wearable Computers (ISWC '99), pp.21-28, October 1999.
13. C. Chen, M. Li, and D. Kotz, “Design and Implementation of a Large-Scale Context Fusion Network,” In. Proc of the 1st Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous’04), pp.246-255, August 2004.
14. K. Fujii and T. Suda, “Dynamic Service Composition Using Semantic Information,” In Proc. of the 2nd International Conference on Service Oriented Computing (ICSOC’04), November 2004.
15. A. Ranganathan, S. Chetan, J. Al-Muhtadi, R. H. Campbell, and M. D. Mickunas, “Olympus: A High-Level Programming Model for Pervasive Computing Environments,” In Proc. of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom’05), pp.7-16, March 2005.
16. Z. Song, Y. Labrou, and R. Masuoka, “Dynamic Service Discovery and Management in Task Computing,” In Proc. of the 1st Annual International Conferences on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous’04), pp. 310-318, August 2004.
17. S. Kalasapur, M. Kumar, and B. Shirazi, “Personalized Service Composition for Ubiquitous Multimedia Delivery,” In Proc. of the 6th IEEE International Symposium on a World of Wireless Mobile and Multimedia Networks (WoWMoM’05), June 2005.
18. D. Karastoyanova, F. Leymann, and A. P. Buchmann, “An Approach to Parameterizing Web Service Flows,” In Proc. of the 4th International Conference on Service Oriented Computing, pp.533-538, December 2005.
19. S. R. Ponnekanti and A. Fox, “Application-Service Interoperation without Standardized Service Interfaces,” In Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communications (PerCom’03), March 2003.
20. R. S. Kaabi, C. Souveyet, and C. Rolland, “Eliciting Service Composition in a Goal Driven Manner,” In Proc. of the 2nd International Conference on Service Oriented Computing (ICSOC’04), November 2004
21. C. Lee and A. Helal, "Context Attributes: An Approach to Enable Context-awareness for Service Discovery," In Proceedings of the Third IEEE/IPSJ Symposium on Applications and the Internet (SAINT 2003), pp.22-30, January 2003.
22. G. M. Voelker and B. N. Bershad, “Mobisaic: An Information System for a Mobile Wireless Computing Environment,” In Proceedings of IEEE Workshop on Mobile Computing Systems and Applications, pp.185-190, December 1994.
23. M. Offermans, “Automatically Managing Service Dependencies in OSGi,” http://www.osgi.org/documents/osgi_technology/AutoManageServiceDependencies_byMOffermans.pdf, May 2003.