Pervasive Services on the Move: Smart Service Diffusion on the OSGi Framework

Davy Preuveneers and Yolande Berbers

Department of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium,
{davy.preuveneers, yolande.berbers}@cs.kuleuven.be,
http://www.cs.kuleuven.be

Abstract. The ubiquity of wireless ad hoc networks and the benefits of loosely coupled services have fostered a growing interest in service oriented architectures for mobile and pervasive computing. Many architectures have been proposed that implement context-sensitive service discovery, selection and composition, or that use a component-based software engineering methodology to facilitate runtime adaptation to changing circumstances. This paper explores live service mobility in pervasive computing environments as a way to mitigate the risk of disconnections during service provision in mobile ad hoc networks. It focuses on context-aware service migration and diffusion to multiple hosts to increase accessibility and expedite human interaction with the service. We analyze the basic requirements for service mobility and discuss an implementation on top of OSGi. Finally, we evaluate our approach to service mobility and illustrate its effectiveness by means of a real-life scenario.

1 Introduction

The late Mark Weiser [1] predicted that the next wave in the era of computing would be outside the realm of the traditional desktop. Everything of value would be on the network in one form or another with smart interacting objects adapting to the current situation without any human involvement. Ubiquitous computing will become the next logical step to mobile computing where people are already connected anytime and anywhere. The growing presence of WiFi and 3G wireless Internet access and sensor network technologies will give rise to this new paradigm, in which information and intelligent services are invisibly embedded in the environment around us.

Service-oriented computing (SOC) [2] is a key enabler of Weiser's vision. It represents the current state of the art in software architecture [3] that utilizes services as fundamental building blocks for the rapid development and deployment of applications. It relies on a service-oriented architecture (SOA) to organize loosely coupled services with functions to bind them and manage their life-cycle, such as the deployment and updating of services. These key features are crucial for service interaction in a ubiquitous computing environment. By breaking up an application into a configuration of independent services with well-defined

interfaces, SOA enables the creation of new applications with the required flexibility to customize services to changing demands and different contexts.

The focus of this paper is on how we can improve the availability and accessibility of services in a highly dynamic ad hoc network where intermittent network connectivity can disrupt an application when multiple services on mobile and stationary devices are temporarily coupled to behave as one single application. We propose to replicate the same service and its state at runtime (i.e. live mobility of services) on multiple devices near to the user that can provide a guaranteed quality of service (QoS) with support for state synchronization between the replicated services. This way, we increase the number of opportunities a user can interact with a service and we can better deal with volatile network connections by handing over to a replica if the connection between two services breaks down. As such, we say that the same service has *diffused* to multiple hosts. In order to keep this diffusion approach scalable, the targets selected for diffusion need to be well-chosen. That is why context-awareness [4] is a necessity for service migration and diffusion in the ubiquitous computing paradigm. It addresses the inner characteristics of the services by collecting relevant information about the service requirements and the devices in the vicinity. It helps to select appropriate targets for service mobility and coordinate synchronization and access to services deployed on these devices, and learns for each service the devices the user will most likely interact with. The applicability of smart service diffusion in a distributed setting will be illustrated on top of the OSGi framework [5].

The paper is structured as follows. In section 2 we list the requirements to enable seamless service migration and diffusion. Section 3 provides details on how we realized these requirements on top of the OSGi framework. In section 4 we conduct experiments that illustrate the effectiveness of service diffusion in a real-life scenario. We measure the overhead of state transfer and synchronization as well as any benefit in improved availability and accessibility. Section 5 provides an overview of related work on distribution and mobility of services. We end with conclusions and future work in section 6.

2 Requirements for service migration and diffusion

Pervasive services offer a certain functionality to nearby users. They are accessed in an *anytime-anywhere* fashion and deployed on various kinds of mobile and stationary devices. When users or devices become mobile, proactive live service mobility to multiple hosts can provide a solution to the increased risk of disconnecting remote services. In this section we review several non-functional concerns and requirements that need to be fulfilled to ensure that the migration and diffusion of a service in a mobile and pervasive setting can be accomplished.

2.1 Device, service and resource awareness

In a pervasive services environment, the multi-user and multi-computer interaction causes a competition for resources on shared devices. Therefore, knowledge
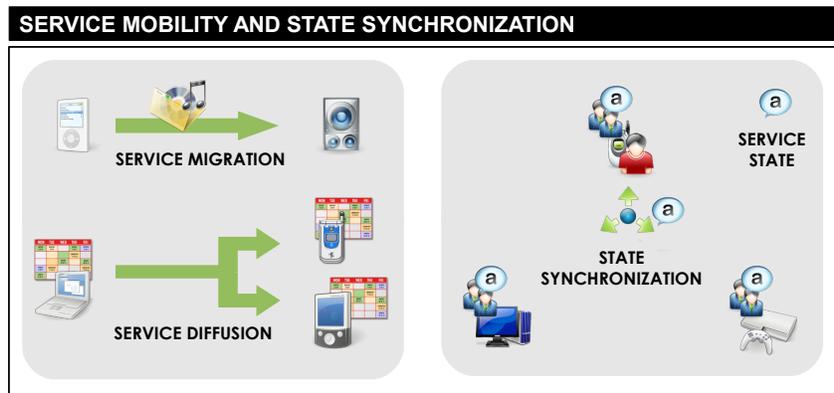
Fig. 1. Live migration and diffusion of services. Service migration completely relocates the service to another host, while service diffusion redeploys the same service on other hosts and ensures service state replication and synchronization.

about the presence, type and context of the devices (including resource-awareness about the maximum availability and current usage of processing power, memory, network bandwidth, battery lifetime, etc.) is a prerequisite to guarantee a minimum usability and quality of service. Before relocating a service to another host, a service discovery protocol (SDP) can be used to verify if the service is not already available [6].

2.2 Explicit service state representation and safe check-pointing

Stateless services can be replaced with similar ones or redeployed on another host at runtime without further ado as long as the syntax and semantics of the interfaces remain the same such that the binding can be reestablished. Stateful services require a state transfer after redeployment before they can continue their operations on a different host. Furthermore, the service must be able to resume from the state it acquired. Therefore, services should model the properties that characterize their state and make sure that check-points of their state are consistent [7, 8].

2.3 State synchronization and support for disconnected operation

Due to possible wireless network disruptions in the mobile setting of the user, complete network transparency of remote service invocations can have a detrimental effect on service availability and accessibility. The service platform should provide support to deal with intermittent network connectivity in order to recover from temporary failures in network connectivity between two services. With handover to replicated services whose states are synchronized, the effect of network disruptions can be reduced. Moreover, with discrete state synchronization, the requirement for continuous network connectivity can be mitigated.

2.4 Enhanced service life cycle management

The service platform must provide functions to manage the life-cycle of services, such as deployment, withdrawal, starting, stopping and updating. If a service cannot be run locally because of a lack of resources, the service needs to be moved to a remote and more powerful device in the vicinity of the user. The service life cycle management should move services to new hosts and replicate the state if necessary. We propose two kinds of service mobility (see Fig. 1):

- **Service Migration:** Once the replicated service has moved to a new host, the original one is stopped and uninstalled. This is the typical behavior of 'follow-me' applications that move along with the user.
- **Service Diffusion:** In this adaptation, the original service becomes active again once the state has been extracted. New replicas acquire the service state and get activated. After activation, service states are synchronized.

Of course, a combination where a service is replicated on multiple hosts, but where the original is uninstalled is also possible. Also note that plain service migration does not require any state synchronization at all, but handing over to a service replica is no longer possible when trying to recover from a network failure. As the effect of network disruptions cannot be mitigated completely, we must also clean up stale replicated services. This problem is similar to distributed garbage collection where heartbeats or lease timeouts are used to detect disconnections and recycle memory. Similar algorithms can be reused to clean up these services.

3 Context-aware service mobility on the OSGi framework

In this section we will discuss how the previous requirements have been implemented on top of the OSGi framework [5]. This lightweight service oriented platform has been chosen for its flexibility to build applications as services in a modularized form with Java technology. OSGi already offers certain functionalities that are needed to implement the service migration and diffusion requirements. OSGi is known for its service dependencies management facilities and the ability to deal with a dynamic availability of services. Moreover, OSGi can be deployed on a wide range of devices, from sensor nodes, home appliances, vehicles to high-end servers, and allows the collaboration of many small components, called bundles, on local or remote computers. As such, OSGi is a viable platform for service orientation in a ubiquitous computing environment.

3.1 Extending service descriptions with deployment constraints

Services in the frame of OSGi are published as a Java class or interface along with a set of service properties. These service properties are key-value pairs that help service requesters to differentiate between service providers that offer services with the same service interface. Service providers and requesters are packaged into a OSGi bundle, i.e. a JAR file with a manifest file that contains

information about the bundle, such as version numbers and service dependencies. Service descriptions are stored in the service registry of OSGi. Service requesters can discover and bind to a service implementation by actively querying for the service or by subscribing to a notification mechanism in order to receive events when changes in the service registry occur.

A recent addition to the OSGi R4 framework that simplifies the registering of POJOs (plain old Java objects) as services and the handling of service dependencies is the Declarative Services (DS) specification [5]. Dependency information that is currently not mentioned in the service descriptor deals with non-functional properties such as hardware, software and resource constraints. For example, one OSGi bundle could be successfully deployed on a J2ME CDC Foundation Profile, while another could need at least a J2ME CDC Personal Profile or a J2SE virtual machine because it uses AWT for a graphical user interface. Moreover, resource (memory, processing power, storage) and hardware (screen, audio, keyboard) dependencies need to specified as well. The current key-value pair format for service properties is too limited to describe these complex constraints. As discussed in our previous work [9–11], ontologies provide a convenient richer specification format to describe and discover pervasive services with support for QoS and context-awareness. We therefore opted to add a 'deployment' entry into the Declarative Service descriptor in which we refer to an ontology that is included in the JAR file of the OSGi bundle:

```xml
<?xml version="1.0"?>
<component name="jabber">
    <implementation class="communication.impl.JabberChatClient" />
    <service>
        <provide interface="communication.ChatClient" />
    </service>
    <deployment>
        <require name="resources1"  class="descriptor.owl#MemoryDependency" />
        <require name="software1"  class="descriptor.owl#JavaVMDependency" />
        <require name="hardware1"  class="descriptor.owl#DisplayDependency" />
        <require name="hardware2"  class="descriptor.owl#KeyboardDependency" />
    </deployment>
</component>
```

It models the above dependencies as class restrictions on concepts defined in our context ontologies[1]. A device should comply with these constraints if the service is to be deployed successfully. For example:

```xml
<owl:Class rdf:about="#MemoryDependency">
  <rdfs:subClassOf rdf:resource="http://www.cs.kuleuven.be/~davy/ontologies/2008/01/
    Hardware.owl#RAM" />
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="http://www.cs.kuleuven.be/~davy/ontologies/2008/01/
        Hardware.owl#currAvailable" />
      <owl:someValuesFrom rdf:resource=">= 98304" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

---

[1] See `http://www.cs.kuleuven.be/~davy/ontologies/2008/01/` for the latest revision of our context ontologies.

The previous constraint declares that a device should have at least one memory resource instance (of class *RAM* or one of its subclasses) with a property *currAvailable* that has a value larger than or equal to 98304. The type of the property is specified in the *Hardware.owl* ontology, and for this value it is *bytes*.

```
<owl:Class rdf:about="#JavaVMDependency" >
  <rdfs:subClassOf rdf:resource="http://www.cs.kuleuven.be/~davy/ontologies/2008/01/
    Software.owl#VirtualMachine" />
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="http://www.cs.kuleuven.be/~davy/ontologies/2008/01/
        Software.owl#hasRenderingEngine" />
      <owl:someValuesFrom rdf:resource="http://www.cs.kuleuven.be/~davy/ontologies/2008/
        01/java.owl#JavaAWT" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The above constraint declares that a device should have at least one Java virtual machine instance with a GUI rendering engine instance that belongs to the JavaAWT class. The main advantage of our context ontologies [9] is that complex semantic relationships only need to be specified once and can be reused by every service descriptor. Moreover, each device profile can specify their characteristics with the same ontologies. If a device then would specify that it runs OSGi on top of JDK 1.6, then an AWT rendering engine dependency would semantically match, but a Java MIDP 2.0 LCDUI rendering engine would not. Resource and hardware constraints can be expressed in a similar way. The matching itself to verify that a service can move to a particular host is carried out by a context enabling service that is described in the following section.

3.2 Context-awareness as an OSGi distributed declarative service

In order to diffuse OSGi services in a way that takes into account the heterogeneity and dynamism of a ubiquitous computing environment, we need to be aware of what the characteristics and capabilities of these devices are, where they are located, what kind of services they offer and what resources are currently available. The COGITO service will provide this information on each host. It gathers and utilizes context information to positively affect the provisioning of services, such as the personalization and redeployment of services tailored to the customer's needs. The core functions of the enabling service are provided as a set of OSGi bundles that can be subdivided into the following categories:

- **Context Acquisition:** These bundles monitor for context that is changing and gather information from resource sensors, device profiles and other repositories within the system itself or on the network.
- **Context Storage:** A context repository ensures persistence of context information. It collects relevant local information and context that remote entities have published in a way that processing can be handled efficiently without losing the semantics of the data.
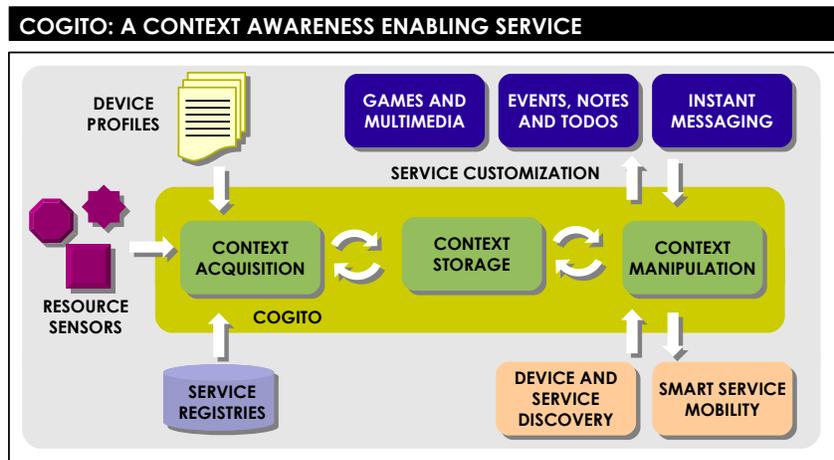
Fig. 2. Building blocks of the COGITO enabling service.

– **Context Manipulation:** These bundles reason on the context information to verify that context constraints are met. Besides a rule and matching engine that exploits semantic relationships between concepts [11], it also provides adapters that transform context information into more suitable formats.

A high-level overview of the building blocks of the context-awareness enabling service is given Figure 2. The advantage of decomposing the context-awareness framework into multiple OSGi bundles is that the modular approach saves resources when certain bundles are not required.

COGITO is implemented as a distributed Declarative Service. As the Declarative Service specification does not cover the registration of remote services, we have each device announcing its presence and that of its context enabling service with a service discovery protocol like UPnP, SLP or ZeroConf. Upon joining a network, each other node in the network creates a remote proxy to the enabling service of the joining node, and the Declarative Services bundle will ensure lazy registration and activation of the proxy whenever remote context is acquired. When a node leaves the network, the proxies on the other nodes are deactivated. This approach for distributed Declarative Services simplifies the collection of context information on the network, but more importantly it also enables transparent sharing of intensive context processing services (such as a rule or inference engine bundle) on resource-constrained devices.

3.3 Service state representation, extraction and synchronization

In order to replicate stateful services on multiple hosts with support for state synchronization, we need to explicitly model the state variables of the service, extract them at runtime and send them to the replicated peers. In order to keep

Fig. 3. New service states in the life-cycle of a migrating or diffusing service. Some of the transitions of the original service are shown in green, the ones of the replicated service(s) in red.

state representation and exchange straightforward and lightweight, we encapsulate the state of a service in a JavaBean. JavaBeans have the benefit of (1) being able to encapsulate several objects in one bean that can be passed around to other nodes, (2) being serializable and (3) providing get and set methods to automate the inspection and updating of the service state. The approach is quite similar to the stateful session beans in the Enterprise JavaBeans specification [12]. Stateful session beans maintain the internal state of web applications and ensure that during a session a client will invoke method calls against the same bean in the remote container. In our approach however, a JavaBean is situated locally within each replicated service and does not involve remote method calls. Instead, the contents of the JavaBean is synchronized with those of the replicated services.

Our current implementation tries to synchronize as soon as the state of an application has been changed. When network failures arise, the state updates are put in a queue and another attempt is carried out later on. The queue holds the revision number of the last state that was successfully exchanged and this for the three replicated applications. If, however, two or more replicated services independently continue without state synchronization, a state transfer conflict will occur when the involved hosts connect again. In that case, a warning will be shown and the user can choose to treat the services as separate applications, or have one service push its state to the others. This approach is rather crude, but acceptable as long as we are mainly dealing with single-user applications.

3.4 Extending the life-cycle management to relocate services

In a ubiquitous computing environment where pervasive services are replicated, a user may switch from one replicated service to another. Therefore, we add extra information to the service state that helps to coordinate the synchronization of the state changes. For example, we add a revision number that is increased after each write operation on the bean and use Vector Clocks [13] among all peers to order the state updating events.

Fig. 3 shows that an active service can not only be stopped, but that in another transition the state can be extracted from the service and synchronized, or that the service can continue to either migrate or replicate to a new host. We use a lease timeout algorithm to garbage collect stale replicated services in order to recycle resources. The timeouts are application dependent, but can be reconfigured by the user.

4 Experimental evaluation

The usefulness and feasibility of dealing with context-aware service migration and diffusion will be illustrated with three applications: (1) a grocery list application, (2) a Jabber instant messaging client, and (3) a Sudoku game. The devices used in the experiment include two PDAs, a laptop and a desktop, all connected to a local WiFi network. This setup will simulate the real-life scenario in the next section.

4.1 Scenario

The scenario goes as follows: *It is Saturday morning and mom and dad go shopping. Everybody at home uses the grocery list application to make up a shared grocery list. In the past, mom sometimes forgot her grocery list at home, but now she only needs to take along her smartphone. The application just diffuses along. Dad also has his PDA with him and runs the same replicated application. They decided to split up the work and go shopping separately. At the mall, each time mom or dad finds a product that they need, it is removed from the shared sticky note application. They get each others updates when the state of the application is synchronized.*

Mom is still checking out while dad is already at the car. As he is waiting, he has another go at the Sudoku puzzle that he was trying to solve on the home computer that morning. His daughter is online and starts a conversation. She wants to send him a picture, but the display of his PDA is too small. He takes his briefcase, turns on his laptop, the application migrates and he continues the conversation there. Unfortunately, he forgot to charge his battery so after a while his laptop gives up on him. "Back to the PDA but then without the picture", dad decides. In the meantime, mom has arrived and dad tells her about the picture. She will have a look at it at home once the application synchronizes with the desktop computer.
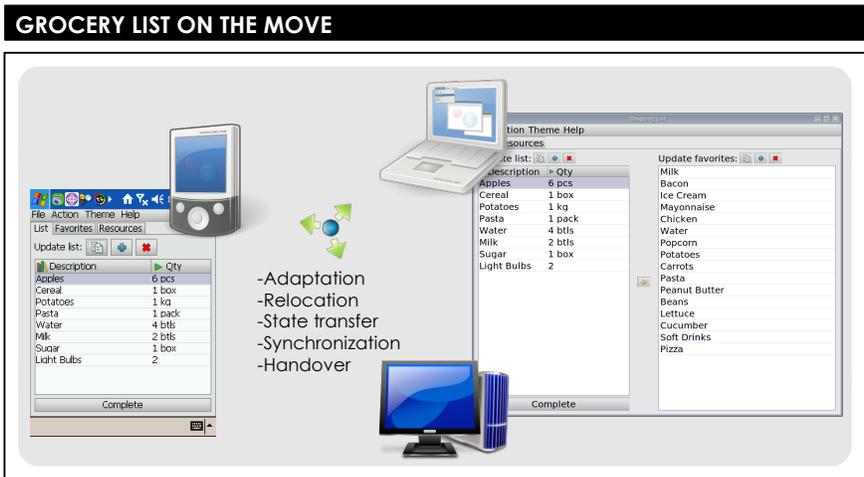
Fig. 4. The grocery list is automatically adapted to fit the new screen size if needed. After redeployment the states of all the replicated grocery lists are synchronized.

4.2 Experiments

In the experiment all the applications are launched on the desktop PC. The Jabber instant messaging client and Sudoku puzzle diffuse to one handheld, while the grocery list application diffuses to both PDAs. The application states are continuously synchronized between all hosts. Later on, the instant messaging application migrates from one PDA to the laptop and back (while synchronizing with the desktop PC). The mobility of the grocery list is illustrated in Figure 4.

Additionally, these three applications all make use of two extra general-purpose services: (i) a stateless audio-based notification service, and (ii) a stateful logging service. By only deploying them on the desktop computer and synchronizing them on the laptop, we enforce that these two services can only be reached from a PDA through a remote connection. The applications will invoke them when (1) the grocery list changes, (2) the online status of somebody changes or (3) when the puzzle is solved. The bindings to the remote services are used to test handover to a replicated service after network disruptions. We let the setup run for 30 minutes, in which the following actions are simulated:

- Each 30 seconds the grocery list is modified and synchronized. This event also triggers an invocation to both the remote services.
- Each 5 seconds the state of the Jabber client is synchronized. Each minute the online status of somebody in the contact list changes, and this triggers an invocation to the remote services.
- The Sudoku puzzle changes each 15 seconds and the 81 digits of the puzzle are synchronized. A puzzle is solved after 10 minutes, and this initiates an invocation to the remote services.

Table 1. Overhead results of state transfer, synchronization and handover.

	Grocery List	**Jabber Client**	**Sudoku Puzzle**
Bundle size	23023 bytes	288235 bytes	54326 bytes
State size	7024 bytes	18235 bytes	1856 bytes
State synchronization	53 Kbytes / min	112 Kbytes / min	37 Kbytes / min
Relocation time	9 seconds	13 seconds	7 seconds
Synchronization delay	350 msec	518 msec	152 msec
Handover delay (audio service)	47 msec	61 msec	54 msec
Handover delay (log service)	167 msec	213 msec	78 msec

Each 3 minutes we shut down a random network connection for 1 minute to verify if the replicated applications can recover from the missed state updates and that handover to one of the remote replicated services. We measured the overhead of state transfer and synchronization and compare that to the resources that the applications required. We also measured the service availability and responsiveness by measuring the delay of handing over to another remote replicated service after a network disconnection. The experiment was repeated 3 times and the averaged results are shown in Table 1.

4.3 Discussion of the results

The test results show there is only a small overhead for the state transfer. This is mainly a consequence of the size of the applications. They are rather small as they would otherwise not run on mobile devices. For other data intensive applications, such as multimedia applications, we expect that the overhead will be a lot bigger if data files have to be moved along as part of the state of the service. As the overhead was limited, we did not implement incremental state transfers, but for media applications this could be an improvement.

The delays in state synchronization are low because the experiments were carried out on a local WiFi network. Other tests in multi-hop networks showed longer but still acceptable state synchronization delays (at most 5 times longer). More interesting though is the difference in service relocation time and state exchange time. By proactively moving the service in advance, the time to get an up-to-date replica is a lot smaller for state synchronization compared to service relocation. As such, service diffusion with state synchronization provides a much better usability to the user compared to plain service migration. Of course, this assumes the required bandwidth is available.

The handover to the remote services worked in all cases because we avoided the case where all network connections to the remote replicated services are disrupted. In theory, our framework can handle this particular case if the remote

service calls can be buffered and invoked asynchronously. However, our current implementation does not support this. Interesting to note for the handover to the replicated services (in our experiment, the stateless audio service and stateful log service), is that handover to another log service takes a bit longer on average. This result is due to the fact that for the audio service we do not need to wait until the replicated service has acquired the latest revision of the service state. If state synchronization is taking place during handover, the handover delay can become higher. In our experiment, the remote services were only available on two hosts (the laptop and the desktop). If more devices would host a replicated service, the decision to handover to a particular device could depend on which replica is already completely synchronized.

5   Related work

In recent years, many researchers have addressed the issue of state transfer, code mobility and service oriented computing in ubiquitous computing environments. This research has greatly improved our knowledge of how context-aware service oriented architectures can accommodate to new functionality requirements in changing environments. Providing a detailed review of the state of the art on all of these domains is beyond the scope of this paper. Instead, we focus on those contributions that are most related to the work presented in this paper.

In [14], the OSGi platform was used to perform context discovery, acquisition, and interpretation and use an OWL ontology-based context model that leverages Semantic Web technology. This paper has inspired us to use the OSGi platform for context-aware service orientation, and more specifically service mobility. In [15], the authors present an OSGi infrastructure for context-aware multimedia services in a smart home environment. Although the application domain that we target goes beyond multimedia and smart home environments, we envision that more data driven services like multimedia applications are good candidates to further evaluate our service mobility strategies. Other context-awareness systems leveraging the OSGi framework include [16, 17].

Optimistic replication is a technique to enable a higher availability and performance in distributed data sharing systems. The advantage of optimistic replication is that it allows replicas to diverge. Although users can observe this divergence, the copies will eventually converge during a reconciliation phase. In [18], the authors provide a comprehensive survey of techniques developed to address divergence. Our current service replication and state synchronization is more related to traditional pessimistic replication, because user perceived divergence might cause confusion. As discussed in [19], we will further investigate how optimistic replication can improve the quality of service in mobile computing, while keeping the user perceived divergence at an acceptable level.

Remote service invocation on the OSGi framework is an issue that has been addressed by several researchers and projects. The Eclipse Communication Framework [20] provides a flexible approach to deal with remote services and service discovery on top of the Equinox OSGi implementation. Remote OSGi

services can be called synchronously and asynchronously on top of various communication protocols. The R-OSGI [21] framework also deals with service distribution. While both projects address network transparency for OSGi services, neither of them deals with explicit service mobility.

The Fluid Computing [22] and flowSGI [23] projects explicitly deal with state transfer and synchronization in a similar way to ours. They also discuss state synchronization and provide a more advanced state conflict resolution mechanism. Our contribution builds upon this approach and specifically focuses on service oriented applications. Our method also enhances the peer selection for service replication by using context information of the service and the device. This approach provides better guarantees that the application will actually work on the device.

6 Conclusions and Future Work

This paper presents a context-driven approach to live service mobility in pervasive computing environments. It focuses on service migration and diffusion to multiple hosts to increase accessibility and expedite human interaction with the service. We summarized the basic requirements for service mobility, including enhanced service descriptions, context-awareness to select appropriate targets for service migration, and life-cycle management support to carry out service relocation with state transfer and state synchronization. We have discussed how we implemented these requirements on top of the OSGi framework and validated our prototype by means of several applications that are replicated in a small scale network with enforced network failures.

We studied the effects of service migration and service diffusion. Service migration moves an application from one host to another including its state, while service diffusion replicates the service and the state on multiple hosts. State synchronization ensures that each service can be used interchangeably. Experiments have shown that the overhead of state transfer and synchronization is limited for relatively small applications. The results illustrate that, if the available network bandwidth permits, the time to keep the state in sync is a lot smaller than to migrate a service from one host to another. This means that if a user wants to use another device because it provides a better quality of service (e.g. the bigger screen in the scenario), that proactive service diffusion provides a much better usability (i.e. shorter delays).

Future work will focus on a better handling of state synchronization conflicts and improving the state encapsulation approach to deal with incremental state transfers for data intensive applications. However, our main concern will always be to keep the supporting infrastructure flexible and lightweight enough for low-end devices with room left to actually run applications. The outcome of this future work could result in a better policy with respect to on how many remote devices a service should be replicated and under which circumstances service migration would be a better approach compared to service diffusion.

References

1. Weiser, M.: The computer for the 21st century. Scientific American **265** (1991) 94–104
2. Papazoglou, M.P., Georgakopoulos, D.: Service oriented computing. Commun. ACM **46** (2003) 24–28
3. Shaw, M., Garlan, D.: Software architecture: perspectives on an emerging discipline. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1996)
4. Dey, A.K.: Understanding and using context. Personal Ubiquitous Comput. **5** (2001) 4–7
5. Open Services Gateway Initiative: OSGi Service Gateway Specification, Release 4.1 (2007)
6. Helal, S.: Standards for service discovery and delivery. Pervasive Computing, IEEE **1** (2002) 95–100
7. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng. **16** (1990) 1293–1306
8. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: An alternative to quiescence: Tranquility. In: ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance, Washington, DC, USA, IEEE Computer Society (2006) 73–82
9. Preuveneers, D., den Bergh, J.V., Wagelaar, D., Georges, A., Rigole, P., Clerckx, T., Berbers, Y., Coninx, K., Jonckers, V., Bosschere, K.D.: Towards an extensible context ontology for Ambient Intelligence. In Markopoulos, P., Eggen, B., Aarts, E., Crowley, J.L., eds.: Second European Symposium on Ambient Intelligence. Volume 3295 of LNCS., Eindhoven, The Netherlands, Springer (2004) 148 – 159
10. Mokhtar, S.B., Preuveneers, D., Georgantas, N., Issarny, V., Berbers, Y.: EASY: Efficient SemAntic Service DiscoverY in Pervasive Computing Environments with QoS and Context Support. Journal Of System and Software **81** (2008) 785–808
11. Preuveneers, D., Berbers, Y.: Encoding semantic awareness in resource-constrained devices. IEEE Intelligent Systems **23** (2008) 26–33
12. Burke, B., Monson-Haefel, R.: Enterprise JavaBeans 3.0 (5th Edition). O'Reilly Media, Inc. (2006)
13. Mattern, F.: Virtual time and global states of distributed systems. (In: Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms)
14. Gu, T., Pung, H.K., Zhang, D.Q.: Toward an osgi-based infrastructure for context-aware applications. Pervasive Computing, IEEE **3** (2004) 66–74
15. Yu, Z., Zhou, X., Yu, Z., Zhang, D., Chin, C.Y.: An osgi-based infrastructure for context-aware multimedia services. Communications Magazine, IEEE **44** (2006) 136–142
16. Lee, H., Park, J., Ko, E., Lee, J.: An agent-based context-aware system on handheld computers. (2006) 229–230
17. Kim, J.H., Yae, S.S., Ramakrishna, R.S.: Context-aware application framework based on open service gateway. Volume 5. (2001) 200–204 vol.5
18. Saito, Y., Shapiro, M.: Optimistic replication. ACM Comput. Surv. **37** (2005) 42–81
19. Kuenning, G.H., Bagrodia, R., Guy, R.G., Popek, G.J., Reiher, P.L., Wang, A.I.: Measuring the quality of service of optimistic replication. In: ECOOP Workshops. (1998) 319–320

20. The Eclipse Foundation: Eclipse Communication Framework. http://www.eclipse.org/ecf/ (2007)
21. Rellermeyer, J.S., Alonso, G., Roscoe, T.: Building, deploying, and monitoring distributed applications with eclipse and r-osgi. In: eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange, New York, NY, USA, ACM (2007) 50–54
22. Bourges-Waldegg, D., Duponchel, Y., Graf, M., Moser, M.: The fluid computing middleware: Bringing application fluidity to the mobile internet. In: SAINT '05: Proceedings of the The 2005 Symposium on Applications and the Internet (SAINT'05), Washington, DC, USA, IEEE Computer Society (2005) 54–63
23. Rellermeyer, J.S.: flowsgi: A framework for dynamic fluid applications. Master's thesis, ETH Zurich (2006)