Adaptive Web Service Migration

Holger Schmidt[1], Rüdiger Kapitza[2], Franz J. Hauck[1], and Hans P. Reiser[3]

[1] Institute of Distributed Systems, Ulm University, Germany
{holger.schmidt,franz.hauck}@uni-ulm.de
[2] Dept. of Comp. Sciences, Informatik 4, University of Erlangen-Nürnberg, Germany
rrkapitz@cs.fau.de
[3] LASIGE, Departamento de Informática, University of Lisboa, Portugal
hans@di.fc.ul.pt

Abstract. In highly dynamic and heterogeneous environments such as mobile and ubiquitous computing, software must be able to adapt at runtime and react to the environment. Furthermore it should be independent of a certain hardware platform and implementation language.

In this paper, we propose an infrastructure for self-adaptive migratable Web services (SAM-WS) for implementing applications for such environments. A SAM-WS supports stateful migration and adaptation to particular application context by being able to dynamically change the interface, locally available state and implementation in use. Despite adaptation and migration it maintains a unique ID during the whole life time. This allows clients to have a location-independent reference to a specific Web service instance. Although our prototype implementation is based on Apache Axis, the concept can be easily ported to any Web service framework without platform modifications. We provide an example application and performance measurements for different system platforms ranging from a standard device to resource-restricted mobile devices.

Keywords: Web Service, Migration, Adaptation.

1 Introduction

In ubiquitous computing (UbiComp) [1], a large number of small devices are interconnected in a dynamic and ad-hoc fashion. Applications for such devices should be platform independent because of the heterogeneous hardware and system software. They have to be adaptive and reactive to cope with the inherent environment dynamics. This is especially the case for mobile applications that are not attached to a specific device. Thus, a UbiComp infrastructure should provide mechanisms to automatically handle heterogeneity and reduce complexity of handling adaptivity and reactivity in the applications.

Mobile processes are an approach to simplify development of applications that change their interaction patterns and location during lifetime. A developer uses a description language (such as proposed by *Kunze et al.* [2]) to specify the behaviour and the interactions of the application. State-of-the-art infrastructures have limitations in terms of supporting adaptation and handling heterogeneity.

R. Meier and S. Terzis (Eds.): DAIS 2008, LNCS 5053, pp. 182–195, 2008.

Adaptation might be necessary for the local state of a process, the current functionality, and the implementation variant. These concerns need to be adjusted according to runtime environment properties, such as the hardware architecture, operating system, available memory, and local devices. Supporting heterogeneity means that processes have to be able to migrate between heterogeneous nodes, without requiring the developer to manually implement code for converting incompatible data representations. Interoperability between different vendor implementations of the infrastructure calls for using standardised protocols.

In previous work, we proposed the concept of a *self-adaptive* mobile process [3]. It can be seen as an ordered execution of services and is able to adapt itself in terms of state, functionality and implementation to the current context (which is represented by the runtime environment) and to migrate either for locally executing services or for accessing particular context, while maintaining its unique identity. We suggest an implementation of mobile processes with Web service technology on the basis of the model-driven architecture (MDA) [4]: developers specify behaviour and interactions of an application using a self-adaptive mobile process description, which is mapped on a self-adaptive migratable Web service (SAM-WS). In this paper, we focus only on the infrastructure for implementing such SAM-WSes without details on the MDA code generation process.

The novel contribution of our infrastructure is that it combines Web service technology with mobility mechanisms that support adaptation and heterogeneity. The key difference to related work on migratable Web services [5,6] is the support for adaptation to the application context by dynamically changing the service interface, the available state and the implementation while maintaining a persistent Web service identity. Unlike our previous work on context-aware migration of CORBA objects [7], in this paper we propose the use of standard Web service technology as core mechanism, which simplifies interoperability between heterogeneous infrastructures of different vendors and allows disconnected operation. On the basis of our dynamic code loading infrastructure [8], the platform contains a novel dynamic deployment service that allows service migration to machines on which the needed code is unavailable and thus has to be loaded on demand. We support client-transparent migration by providing persistent Web service references for the whole life cycle by introducing a persistent Web service identity. Although our prototype is implemented using Apache Axis, the concept provides a generic life cycle service specification for Web services.

The paper is structured as follows. First, we discuss related work. In Section 3, we present our design of adaptive Web service migration: basic principles, requirements and logical entities. After an in-depth description of our infrastructure in Section 4, we sketch a basic example application and show performance measurements in Section 5. We conclude and draft future work in Section 6.

2 Related Work

There are several projects targeting adaptation and context-sensitivity of Web services. For instance *Erradi et al.* developed a policy-based middleware for

adaptive composite Web services [9]. Adaptation is based on dynamic Web service composition in these systems, wheras we replace the original Web service with an adapted one. Thus, our approach allows optimal resource usage by adapting a Web service to a device-tailored one. The Web service composition approach for adaptation can be used for integrating legacy services in our approach.

There exists a lot of work in the area of object migration, in particular for mobile agents (objects with an autonomous activity). However, many systems, such as *Aglets* [10] rely on native Java serialisation and are therefore restricted to a homogeneous environment and do not support adaptation to the context.

In previous work, we presented a concept for weak object migration on basis of the CORBA life cycle service [11]. For implementing mobile objects we use CORBA value types, i.e., objects with call-by-copy semantics. Thus, the developer does not have to care about externalisation and internalisation of the mobile object as this is handled transparently by the CORBA system. Our implementation does not support adaptation to the current application context.

There also exist systems that support adaptive object migration. For instance, *Almeida et al.* developed a dynamic reconfiguration service for CORBA [12]. The developer has to implement methods for internalising and externalising the object state, which may lead to error-prone implementations.

Recently, we introduced a concept for context-aware object migration on basis of the CORBA life cycle service [7]. To the best of our knowledge, this service is the only one allowing dynamic adaptive object migration regarding state, interface and code at runtime. However, it is restricted to CORBA and does not enable Web service migration and disconnected operation.

Hammerschmidt and Linnemann developed a service for stateful Web service migration [5]. However, as the approach builds on native Java serialisation, it is limited to homogeneous Java environments. The system does not provide concepts for adaptation of the state, interface and code. Furthermore, *Ishikawa et al.* describe a system for supporting Web service integration for pervasive computing [6]. In this approach mobile agents implement workflows. The agents can move to Web service locations in order to obtain efficient local access. However, their system does not support adaptation and is restricted to Java as it uses native Java serialisation.

Our system enables the implementation of mobile workflows (see example in Section 5). However, in contrast to a mobile workflow management system such as proposed by Satoh [13], which transfers documents, our system allows the complete migration of services. This enables tailored adaptation of the application to the current context and an on-demand instantiation of the application on devices that are not aware of the application in advance.

3 Design of Adaptive Web Service Migration

In this section, we first give an overview on basic principles on which our concept for adaptive Web service migration builds. Then, we present requirements that we identified and present necessary logical entities.

3.1 Basic Principles of Self-adaptive Web Service Migration

Web Service migration is the concept of moving a Web service from one machine to another at runtime. This requires transferring the Web service's implementation code. In this paper, we introduce *stateful self-adaptive migratable Web services* (SAM-WS), which have the advantage of supporting dynamic adaptation on the basis of current run-time environment and explicitly specified criteria. Our infrastructure introduces a concept for uniquely referencing a Web service independent of its location on basis of a location tracking service (see Section 4.5). For this purpose, the service URL is augmented with a globally unique ID (GUID). This allows the coexistence of several Web service instances at a particular location. We allow adaptation of the Web service's provided functionality (i.e., interface), the used internal state (i.e., set of variables) and the implementation code. Neither adaptation nor migration influence the GUID of the Web service, which allows continuous identification and addressing of the Web service. The GUID is automatically generated at deployment time.

For implementing self-adaptation, we introduce the concept of *Web service facets*. These represent a particular characteristic of the migratable Web service with a specific configuration of interface, state and implementation. Figure 1 shows a Web service facet providing interface A, using internal state a, b, and c and running an implementation in Java which adapts itself in context of migration to a Web service facet providing interface B, using internal state b, c, d and running an implementation in C++. As mentioned before, it is important to note that the globally unique ID is preserved during migration. The assignment of service state from one Web service facet to another one is realised using a name matching algorithm: If a Web service facet #1 contains state attributes with name `b` and `c`, the states `b` and `c` within another Web service facet #2 are considered the same (see Figure 1). Thus, after adaptation from Web service facet #1 into Web service facet #2, the state `b` and `c` of facet #2 has to be set to the prior state `b` and `c` of facet #1 (type incompatibilities result in an error).

Stateful adaptive Web service migration requires transferring the service state from the source to the target. By enabling a replacement of the implementation upon migration of the Web service, transfer states have to be interpretable by any possible implementation. Thus, we differentiate *implementation-dependent* and *implementation-independent* state of a Web service. We define implementation-independent state as the part of the service state that should be interpretable by any possible implementation of a specific functionality. For example, in

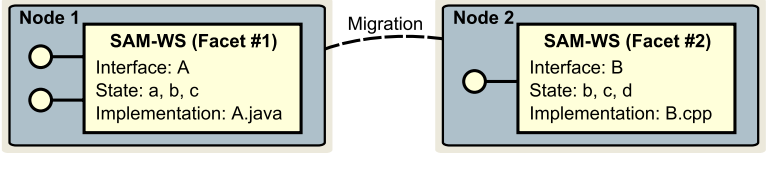

Fig. 1. Adaptive migration from one Web service facet into another one

case of a hash table functionality this would be only the key-value pairs. We consider state such as internal variables of managing structures for the hash function as implementation-dependent state which varies from one implementation to another. In case of migration, such implementation-dependent state can only be interpreted by the same target implementation. Thus, we transfer implementation-independent state only. However, implementation-independent state cannot be automatically determined. Thus, the developer of a SAM-WS has to tag implementation-independent state manually (see Section 4.1).

Additionally, for enabling SAM-WSes, we introduce a differentiation of *active* and *passive service state*. By adapting a Web service to a specific Web service facet, parts of the service state can be left out and other parts can be added. We call leaving out parts of the service state *passivation* and allow a subsequent *activation* of this state within another Web service facet. Therefore, passive state has to be stored for later use (see Section 3.3).

3.2 Requirements for Web Service Migration

We identified several requirements for our infrastructure:

Ubiquitous computing environments are characterised by *high dynamics* and *heterogeneous infrastructure*. Dynamics require supporting run-time decisions, e.g., selecting appropriate migration targets. Additionally, dynamic loading of locally unavailable code should be supported for allowing migration to any possible target location (even if the code is not known there before). The heterogeneous infrastructure requires platform- and language-independent techniques for communication and for state transfer. We think that by building on XML, Web service technology is appropriate for such environments.

Due to the heterogeneous infrastructure, we advocate requiring *self-adaptation* according to the application context. For example, this enables dynamic replacement of the implementation and thus running the same functionality on a mobile device with a lean and restricted implementation as well as on a workstation with a fully-fledged implementation.

For intuitive usage of SAM-WSes these should offer *client-side transparency*. Clients should notice neither migration nor adaptation of the Web service. This requires *continuous addressing* of the migratable Web service for the whole life cycle without client notice. Every SAM-WS should have a service-specific management interface, which every possible Web service facet has to support for providing some kind of stable interface part.

Furthermore, there should be *application development support*. For instance, developers should be offered an interface that provides high-level migration and adaptation support based on criteria, which allow the specification of target locations, context requirements and adaptation requirements.

3.3 Logical Entities and Collaboration

This section provides a brief overview of logical entities for SAM-WS migration. Figure 2 shows the interaction.

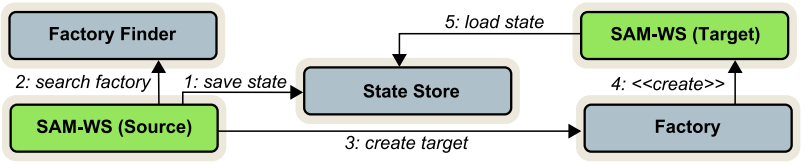

Fig. 2. Collaboration of logical entities for adaptive Web service migration

When a SAM-WS decides to migrate (this could be triggered internally or externally), it has to store the Web service's active state into a *state store* service for later use (passivation, see Section 3.1). To guarantee a consistent state transfer, migration has to be coordinated with request execution (see Section 4.3). Then, the SAM-WS tries to discover possible migration targets with the help of a *factory finder* service. Therefore, the SAM-WS passes criteria to the factory finder service (e.g., required context and provided interface at the target) according to which appropriate *factory* services (i.e., migration targets) are returned. These factory services enable the remote deployment of arbitrary Web services (if code is existent and executable for the particular platform). The factory service allows the creation of the criteria-specified Web service facet at the desired location. Last, the newly created Web service is updated with the necessary state from the state store service, the original Web service is undeployed and references to the Web service are updated to the new location (see Section 4.5).

4 Infrastructure for Adaptive Web Service Migration

In this section, we sketch our infrastructure for supporting SAM-WSes. First, we give details on the adaptive Web service migration process with its compulsory entities within our prototype for Apache Axis[1]. However, our concept is generic and can be applied to other Web service containers as well. Then, we show development steps for implementing SAM-WSes. Furthermore, we present advanced concepts for coordination of migration with request execution, dynamic loading of code and continuously addressing a SAM-WS for its whole life time.

4.1 Process of Adaptive Web Service Migration

Figure 3 shows the collaboration of implementation entities for adaptive Web service migration. However, before migration is processed it has to be coordinated with request execution (see Section 4.3). In the first step, the SAM-WS's `move` method is called[2]. The method can either be called directly by a client or by the SAM-WS itself, which provides a mechanism to enable autonomous behaviour. For specifying the migration target, the URI to the preferred factory finder service as well as (key, value)-pairs of non-functional criteria describing

[1] http://ws.apache.org/axis/
[2] The SAM-WS implements the interface `AWSMService` within our prototype.

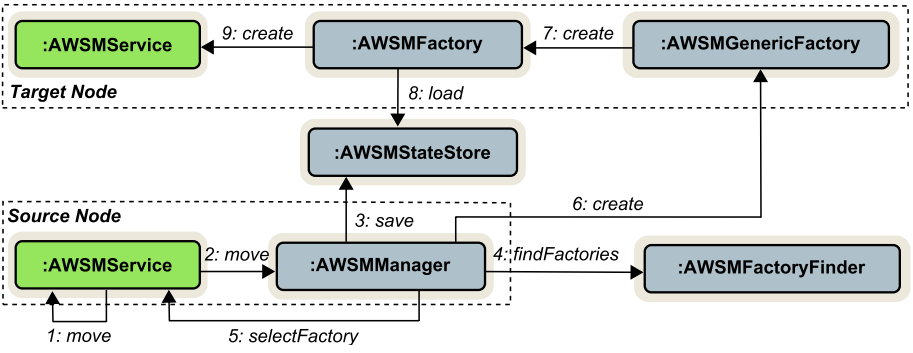

Fig. 3. Collaboration of implementation entities for adaptive Web service migration

appropriate migration targets (e.g., required context and interface) have to be passed as parameters.

AWSMManager. For simplifying development, we provide a local `AWSMManager` Web service, which manages further migration steps. It provides a `move` method that is called by the migrating SAM-WS. As parameters, the SAM-WS passes the given factory finder URI and criteria describing appropriate migration targets. Additionally, a self-reference, which is used for state introspection, and the service ID have to be passed.

Before migrating the SAM-WS, the implementation-independent state has to be extracted. Therefore, this state has to be described within the WSDL description to allow a language-independent specification (see Fig. 4). Within our prototype implementation running in Java we allow annotating implementation-independent state and provide a tool, which automatically generates the WSDL description. On the basis of the WSDL description, the implementation-independent state can be extracted automatically. Then, the state has to be stored (i.e., passivated) to the state store service for future use (see below).

```
1  <wsdl:definitions xmlns:wsdl="...">
2    <wsdl:types>...</wsdl:types>
3    <wsdl:portType name="Test">
4      <wsdl:operation name="getX"> ...
5      </wsdl:operation>
6    <wsdl:service name="TestService">
7      <wsdl:port>...</wsdl:port>
8      <awsm:states xmlns:awsm="...">
9        <state>x</state> ...
10     </awsm:states>
11   </wsdl:service>
12 </wsdl:definitions>
```

Fig. 4. WSDL description with implementation-independent state

```
1 public interface AWSMStateStore {
2   public String getStates(int id, String stateNames[]);
3   public String getStates(int id);
4   public void store(int id, String xmlState);
5 }
```

Fig. 5. Java interface of the `AWSMStateStore` Web service

The infrastructure has to select an appropriate migration target. A factory finder service assists in this selection by, given a list of criteria, returning appropriate factory services as a list (see below). The `AWSMManager` invokes a call-back method at the migrating SAM-WS to support an application-specific selection.

Last, our `AWSMManager` creates and deploys the necessary SAM-WS facet at the new location according to the given requirements, undeploys the original SAM-WS and updates the SAM-WS reference (see Section 4.5).

AWSMStateStore. The `AWSMStateStore` Web service provides methods for storing and retrieving state with respect to a specific service ID. As already mentioned in Section 3.1 this is needed for implementing passive state of the SAM-WS, which is non-existent within a particular facet, but may be used again within another facet. Figure 5 shows the interface of the `AWSMStateStore`. For retrieving only necessary parts of the current Web service facet, the `AWSMStateStore` service provides a custom `getStates` method with a parameter for specifying such parts of the state. For interoperability reasons with other Web service platforms, we use an XML string representation for passing state. This XML state representation is automatically generated within our `AWSMManager` on the basis of the WSDL description containing the implementation-independent state and parsed within the `AWSMStateStore`.

We provide a basic `AWSMStateStore` Web service implementation that internally stores the XML state data in the memory. However, on the basis of the `AWSMStateStore` interface, there can also be more complex implementations, e.g., using a database or peer-to-peer mechanisms for decentrally storing data.

AWSMFactoryFinder. The `AWSMFactoryFinder` implements a kind of factory service repository and represents an abstract service location. Factory services can be discovered as soon as they make an initial registration at the `AWSMFactoryFinder`. The factory finder service has a `register` method, which receives the WSDL-URI of a factory service and a corresponding set of criteria that the factory service provides (see Fig. 6). For interoperability reasons with other platforms, these criteria are transferred as an XML string representation. This data is stored in some kind of factory service repository with provided criteria. For deleting factory services there is an `unregister` method accepting the affected factory service's WSDL-URI as a parameter.

The `AWSMFactoryFinder` service provides an interface with methods for searching for factory services according to given criteria. We allow the specification of required context (e.g., physical/network location, CPU power, memory)

```
1 public interface AWSMFactoryFinder {
2   public void register(String xmlCriteria, String wsdlAddress);
3   public void unregister(String wsdlAddress);
4   public String[] findFactories(String xmlCriteria);
5 }
```

Fig. 6. Java interface of the `AWSMFactoryFinder` Web service

and provided functionality. These capabilities of the `AWSMFactoryFinder` enable two types of Web service migration: *Context-based migration* targets at running the SAM-WS on a platform that provides the desired context and *functionality-based migration* targets at running a specific Web service facet, e.g., for implementing the next step within mobile workflows (i.e., the mobile workflow is implicitly implemented by a SAM-WS, workflow steps are implemented by adaptation to specific Web service facets; see Section 5). Internally, for searching for appropriate factory services, the `AWSMFactoryFinder` selects adequate factory services from its repository and returns the corresponding WSDL-URIs.

We also allow using UDDI for discovery of factory services. In contrast to UDDI, our factory finder service eases the integration of policies according to which factories are returned (e.g., unordered list and best-fitting first).

AWSMGenericFactory and AWSMFactory. The logical factory service entity from Section 3.3 is split into two entities in our prototype implementation. The `AWSMGenericFactory` Web service is responsible for creating a SAM-WS-facet-specific `AWSMFactory` Web service. We need this delegation mechanism for integration of our dynamic code loading infrastructure [8], because it allows loading the `AWSMFactory` code before creation (see Section 4.4). Direct registration of an `AWSMFactory` at the `AWSMFactoryFinder` is possible as well.

The `AWSMFactory` enables dynamic deployment of necessary SAM-WS facets. Therefore, it offers a `create` method, which takes the SAM-WS ID as well as mandatory criteria as parameters. On the basis of the passed criteria, an appropriate SAM-WS facet is selected and instantiated. By using the given ID, the necessary state is retrieved from the `AWSMStateStore` and initialised within the new SAM-WS facet with keeping the original ID. Then, the Web service facet is deployed to allow remote access. Therefore, a deployment descriptor as well as the WSDL interface is generated automatically at runtime if required.

4.2 Development of Self-adaptive Mobile Web Services

For developing a SAM-WS, the developer has to decide which kind of facets a service should offer. Then, she has to implement them for each platform that should be supported, according to these conventions:

- Only implementation-independent state should be considered for migration and adaptation. It has to be marked either by our Java annotation *@ImplementationIndependentState* or within the WSDL file (see Section 4.1)

- Implementation-independent state defined within one SAM-WS facet is mapped to another facet by name matching (see Section 3.1)
- The implementation has to implement the `AWSMService` interface, which provides life-cycle methods of the SAM-WS (`move`, `copy`, `remove`)
- The implementation has to implement the `StatefulService` interface, which provides introspection methods of the SAM-WS (`getState`, `setState`).

For easing development efforts, we provide an abstract `AWSMServiceImpl` class, which contains generic code for introspection (on the basis of annotations and WSDL), generation of the globally unique ID and migration methods. Thus, the developer only has to inherit from this class and to ensure specification of state and state consistency among the different SAM-WS facets.

Then, the developer has to generate standard Web service packages of the SAM-WS facets for the required platforms (e.g., standard Web archive for Apache Axis). These packages are deployed and registered at our dynamic code loading infrastructure for loading these packages on demand (see Section 4.4).

4.3 Coordination

For maintaining consistent state with migration and adaptation, coordination is required. First, migration should only be possible if no other requests are currently handled by the SAM-WS. We use an interceptor at the server side for counting the number of currently active requests. Safe migration is possible if the current migration or adaptation request is the only active. Thus, such a request can only execute as soon as the request counter is equal to 1. As soon as a migration or adaptation is requested, all subsequent requests are deferred. After all previous requests have returned, the migration is started, and after successful migration, the deferred requests are forwarded to the new location.

In our prototype implementation for Apache Axis, we implemented a `SOAPHandler` by extending Apache Axis' abstract class `BasicHandler`. There, an `invoke` method is called with passing a `MessageContext` object from the Apache Axis container for every SOAP request and response. The `MessageContext` object contains the affected service and service method. This allows the sole interception of a specific SAM-WS; otherwise, other Web services would be affected as well. The `SOAPHandler` has to be registered at the container.

4.4 Dynamic Loading of Code

For enabling migration to Web service containers where the necessary code is locally unavailable, we integrated a dynamic code loading service. Dynamic code loading is an essential part of service migration, especially in a dynamic environment without guarantee of local existence of required code.

We developed a decentralised code loading service (P2P-DLS) [8]. It allows any peer to offer and to obtain platform-specific code. We proposed a dynamic loading infrastructure that is independent from the peer-to-peer mechanism in use. Based on our generic concept, we developed a JXTA-based service [14].

For supporting dynamic code loading within our infrastructure, we integrated the P2P-DLS into the `AWSMGenericFactory` (see Section 4.1). The generic factory service queries the P2P-DLS for appropriate location-dependent Web service facet implementations. The `AWSMGenericFactory` service identifies the necessary code by the interface name, and loads this code on demand for instantiating factories, which are specific for deploying a particular Web service facet.

For addressing security issues regarding dynamic code loading standard security mechanisms like code signing could be easily integrated.

4.5 Addressing Self-adaptive Mobile Web Services

Even though a SAM-WS is mobile as well as self-adaptive it can be continuously addressed using the SAM-WS service URL, which also contains the service ID. We implemented a location tracking service that is able to manage current locations of a defined set of SAM-WSes. Therefore, Web services initially register a public service address at the location tracking service and identify themselves using their current service address with the globally unique ID. The public service address, which is located at the location tracking service container, is used as permanent Web service reference; invocations are redirected by the Web service container using the location tracking service data. Whenever a Web service changes its location, it notifies the location tracking service about the new location (i.e., reference is updated). For client-transparency SAM-WSes should implement a management interface being stable within each facet (see Section 3.2).

For improving performance in our prototype for Apache Axis, we implemented an `HTTPRedirector` for client-side interception of SOAP requests over HTTP. This redirector has to be deployed at client-side, which results in every invocation going through the interceptor. Current locations of SAM-WSes, which are given in a redirect response, are cached. Thus, further invocations are directly forwarded without redirection (an error, i.e., a `404 Not Found` response, leads to invoking the original service URL again).

5 Example Application

Our approach provides a basis for the development of flexible and dynamic applications, e.g., for UbiComp. We present a mobile reporter application, in which reporters spontaneously initiate a mobile workflow: reporters enter data into a local Web service, which migrates onto a reviewer's machine for checking the data, and then migrates on a publisher's machine for publishing the content. Reporters become reviewers after a number of accepted reports. This requires dynamic deployment, which is also enforced by the fact that participants may spontaneously join and therefore have to deploy the application on demand.

Such an application can be implemented using SAM-WSes. Web service facets represent different roles within the mobile workflow: reporter, reviewer and publisher facet (see Fig. 7). In contrast to standard workflow systems, the workflow in our system comes along with the code, which can preserve computing resources

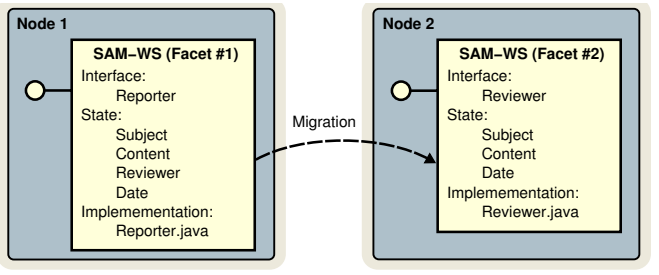

Fig. 7. Self-adaptive migration from reporter facet into reviewer facet

for workflow interpretation on resource-limited devices. Our transparent concept for addressing the SAM-WS enables service observation whenever required.

We measured the time for migrating from the reporter into the reviewer facet. The measurements were performed on an AMD Athlon with 1.73 GHz and 1GB RAM with two Apache Axis 1.4 containers (migration source/target) running on Apache Tomcat 5.5.12 with Java 1.5.0_08. Table 1 shows the overall result and the time for each of the process steps. We measured the performance of 30 Web service migrations and calculated the average time needed.

Overall, self-adaptive migration takes some time; especially WSDL generation as well as deployment are noticeable at the migration target. However, WSDL generation performance can be improved using caching mechanisms. Deployment at the migration target within the Apache Axis container takes around 72% of overall migration time. We are confident that future generations of Apache Axis provide improved deployment performance, which may rigorously improve overall migration time.

For comparison, we measured the time for migration without adaptation of the reporter facet. As migration steps are the same, overall time is comparable: 10229 (±582) ms. The moderately increased migration time compared to Table 1 results from the fact that the complete state is transferred, whereas in case of migration into the reviewer Web service facet the `reporter` state is omitted.

Considering embedded and mobile devices, we have done the same performance measurement for two somewhat outdated ARM-based as well as for a

Table 1. Migration from reporter into reviewer facet on standard device

Migration source (reporter facet)				
State extraction	*State storage*	*Find factories*		*Sum*
3±13 ms	101±103 ms	35±29 ms		139 ms
Migration target (reviewer facet)				
WSDL generation	*State loading*	*Deployment*	*State setting*	*Sum*
2378±58 ms	90±34 ms	7399±32 ms	19±17 ms	9886 ms
				Overall: 10184 ms ±580 ms

Table 2. Migration from reporter into reviewer facet on embedded/mobile devices

Device	CPU	Memory	Migration Time
Embedded System	Strong ARM 233 MHz	256 MB	80±4 s
Handheld (HP Jornada)	Strong ARM 200 MHz	32 MB	230±12 s
Subnotebook (Asus EeePC 4G)	Intel Celeron M 900 Mhz	512 MB	9±0.5 s

current x86-based device (see Table 2). On outdated devices our Apache Axis approach does not perform well, but the measurement on the current device with much more computing power result in better figures. As our concept relies on standard Web service technology, this can be improved even further by optimised Web service containers for small devices.

6 Conclusion and Future Work

In this paper, we proposed a novel infrastructure for self-adaptive migratable Web services. These Web services enable the implementation of UbiComp applications by supporting very flexible adaptation to particular application context (dynamic change of the interface, locally available state and implementation in use). This allows an adaptation of a fully-fledged implementation on a powerful device to a restricted implementation on a resource-limited device. We implemented a prototype for the Apache Axis Web service container. As our system builds on top of standard Web service technology without any modifications, we allow interoperable implementations for other Web service containers as well. However, for supporting coordination and continuous addressing of the SAM-WS clients as well as containers have to support interception of invocations. We prove the feasibility of our approach with a basic reporter example application and performance measurements for different platforms.

For future work, we plan to implement a prototype for another Web service platform. We do not expect interoperability problems, as we designed our infrastructure to only rely on standard Web service technology. For an improved appliance in ubiquitous computing scenarios we will investigate the implementation of our concept using the Java Micro Edition.

Our approach for self-adaptive migratable Web services provides a very flexible concept. This may lead to error-prone applications whenever migrating into unanticipated facets (this may, e.g., result in unavailable state). Therefore, we will examine concepts for defining rules for the specification of allowed migration of Web service facets into other ones. For supporting this specification process we are investigating an MDA-like approach as proposed in our recent work [3].

References

1. Weiser, M.: The computer for the 21st Century. Sci. American 265(3), 66–75 (1991)
2. Kunze, C.P., Zaplata, S., Lamersdorf, W.: Mobile Process Description and Execution. In: Eliassen, F., Montresor, A. (eds.) DAIS 2006. LNCS, vol. 4025, Springer, Heidelberg (2006)

3. Schmidt, H., Hauck, F.J.: SAMProc: Middleware for Self-adaptive Mobile Processes in Heterogeneous Ubiquitous Environments. In: MDS 2007, ACM Press, New York (accepted for publication, 2007)
4. OMG. MDA Guide Version 1.0.1. OMG Doc. omg/2003-06-01 (2003)
5. Hammerschmidt, B.C., Linnemann, V.: Migratable Web Services: Increasing Performance and Privacy in Service Oriented Architectures. IADIS Int. J. on Comp. Sci. and Info. Sys. 1(1), 42–56 (2006)
6. Ishikawa, F., Yoshioka, N., Tahara, Y., Honiden, S.: Mobile Agent System for Web Services Integration in Pervasive Networks. In: IWUC 2004, pp. 38–47 (2004)
7. Kapitza, R., Schmidt, H., Söldner, G., Hauck, F.J.: A Framework for Adaptive Mobile Objects in Heterogeneous Environments. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4276, Springer, Heidelberg (2006)
8. Kapitza, R., Schmidt, H., Bartlang, U., Hauck, F.J.: A Generic Infrastructure for Decentralised Dynamic Loading of Platform-Specific Code. In: Indulska, J., Raymond, K. (eds.) DAIS 2007. LNCS, vol. 4531, Springer, Heidelberg (2007)
9. Erradi, A., Tosic, V., Maheshwari, P.: MASC -.NET-Based Middleware for Adaptive Composite Web Services. In: ICWS 2007, pp. 727–734 (2007)
10. Lange, D.B., Oshima, M.: Programming and Deploying Java Mobile Agents with Aglets. Addison-Wesley, Reading (1998)
11. Kapitza, R., Schmidt, H., Hauck, F.J.: Platform-Independent Object Migration in CORBA. In: Meersman, R., Tari, Z. (eds.) OTM 2005. LNCS, vol. 3760, Springer, Heidelberg (2005)
12. Almeida, J., Wegdam, M., van Sinderen, M., Nieuwenhuis, L.: Transparent Dynamic Reconfiguration for CORBA. In: DOA 2001, IEEE, Los Alamitos (2001)
13. Satoh, I.: Network Processing of Documents, for Documents, by Documents. In: Alonso, G. (ed.) Middleware 2005. LNCS, vol. 3790, pp. 421–430. Springer, Heidelberg (2005)
14. Gong, L.: JXTA: A Network Programming Environment. IEEE Internet Comp. 5(3), 88–95 (2001)