IADIS International Journal on Computer Science and Information Systems
Vol. 1, No. 1, pp. 42-56
ISSN: 1646-3692

MIGRATABLE WEB SERVICES: INCREASING PERFORMANCE AND PRIVACY IN SERVICE ORIENTED ARCHITECTURES

Beda Christoph Hammerschmidt andVolker Linnemann
Institute for Informations Systems
University of Lübeck, Germany

ABSTRACT

Common web service architectures follow the classical client-server model with the client bound to the web service by a static physical connection. In this paper we show that this model is too restricted for some business scenarios and motivate the paradigm and the advantages of *migratable web services.* Migratable web services are instances of conventional web services that can change their executing host at runtime without loosing the actual state and the connection to their clients. Migratable web services exceed remote installation of code because the current state of a web service instance is preserved.
We present a prototypical implementation based on *Apache Axis* which allows the seamless migration of arbitrary web service instances between different hosts. The connection to the clients is not affected by migration processes as the physical client-server model is abstracted to a logical client server model. The discovering of migrated service instances may use centralized as well as decentralized approaches. We present a *JXTA* based P2P grid that is used to discover an instance of a web service after multiple unnoticed migrations.

KEYWORDS

Web Services, Migration, Mobile Agent Systems, Service Oriented Architecture

1. INTRODUCTION

Web Services have continuously gained importance in business and research in the last years. With XML (SOAP) based communication between web services and their clients the web service paradigm is almost independent of platforms, operating systems and programming languages. Discovering and binding of web services at runtime lead to a flexible and dynamic architecture. Although web services follow the classical client-server model the client application has to discover the physical location (URIs) of relevant web services and bind to them. This look-up operation can be done using registries like the *Universal Description, Discovery and Integration* (UDDI) [15]. For the rest of the processing between client and service the connection remains static. Although UDDI supports the dynamic binding of a client to a web service, it does not distinguish between different instances of the same web service.

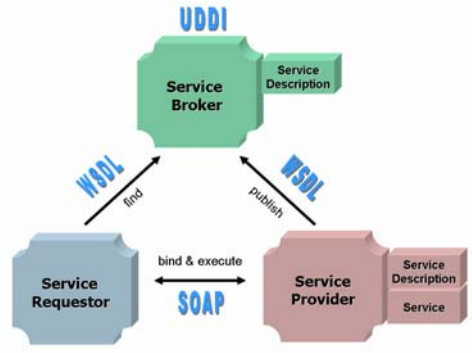

Figure 1. The web service triangle

In some (business) scenarios this architecture is too inflexible and does not comply with business relations in the real world. In this paper we extend the web service paradigm by allowing stateful instances of a web service to roam (migrate) between different hosts. We present three sample scenarios to motivate the need for these so called *migratable web services*:

Example 1: Data Protection:

Imagine an online shop with several customers having unvalidated addresses. The shop owners want to verify the data pool by checking the zip codes and the correctness of names of cities and streets. Let us assume that there are web services available that are able to perform this task. With conventional web services the data of customers are sent to the web service's host, processed there and the result is sent back to the client. For personal and valuable data it is undesired or even prohibited by law to send it over the Internet. Security related approaches like *Web Service Security* (WSS) [16] guarantee an encrypted communication between client and web service that cannot be tapped or manipulated. But in all cases the client has to trust the web service that decrypts and processes the data. This fact prevents the client of using unknown web services leading to restricted and inflexible data processing.
This conflict can easily be solved using the paradigm of *migratable web services*: after being bound by the client, the web service is migrated to a trustworthy host in the client's environment. A security mechanism restricting the access of executed code to resources prevents a malicious web service like a trojan from connecting other hosts and sending secret data. Examples include the *Java Sandbox* [26] or more sophisticated security approaches, e.g. [32]. Using migratable web services a client is able to use even unknown web services without risking the loss of data security.

Example 2: Client-centric Performance / Bandwidth Reduction:

A web service with a few or moderate size of code that typically processes huge data benefits from being migrated to the clients host where the data is accessible locally. An example may be a web service processing images with filter operations. In this case a web service instance is parameterized by the client's demands and gets a raw image embedded in the requests of the client in order to return it after the processing. It is obvious that the huge amount of data that is accompanied by image processing requests a high bandwidth. The response time for the client's requests is enormous because the transportation of data takes the majority of time. The migration of the web service to the client with the images stored locally will decrease the processing time as only the code of the web service has to be transmitted once. Additionally,

the web service provider needs less computational power as the web services are executed on the clients' machines.

Example 3: Server-centric Performance / Load Balancing:

A provider that hosts highly requested web services regularly holds a cluster of hosts to execute numerous instances of web services simultaneously. An example for this scenario might be a business-to-business application that is used by a popular book shop or a travel agency. This application is executing a multitude of client requests at the same time. Different instances of the same web service are executed on several hosts which have to be dimensioned performant enough to process multiple requests at the same time. Therefore the maximal power of each host has to exceed its average load leading to higher total operation costs.
Using migratable web services, the provider is capable to redistribute the web service instances to less loaded hosts without affecting the clients and their connection to the service. In consequence, the provider needs less server capacity to process the client's requests and the clients benefit from an improved response time.

On the first look it seems that the mentioned problems can be solved using remote code installation (RCI). With RCI the code of a web service is copied and deployed on another host. But as RCI ignores the actual state of a web service instance it is not suitable for personalized or parameterized services. A personalization of a web service may be the image processing parameters in example two, for instance. The state depends on the client and may include information like payment information, rights, client specific parameters, etc. If the web service's behaviour depends on a previously submitted login of the client, we can speak of a personalized web service. One common approach to implement personalized web services is to create one separate instance of the web service for each active client (comparable to the Session scope of Java Servlets or Java Server Pages). The life cycle of an instance begins with the login and ends after processing all client orders. A frequently requested web service will lead to a multitude of instances with different states running simultaneously. RCI still follows the client-server model: if a web service is copied and deployed to another host the clients are not redirected automatically. In our opinion, RCI is sufficient for stateless web services offering a limited scope without continuous interactions with the client.
The remainder of this paper is organized as follows: the paradigm of migratable web service is defined in Section 2. In Section 3 we present details of our implementation which is based on Apache Axis and the JXTA P2P framework. In Section 4, we give an outline about related work starting with environments for mobile web services and agent systems. With an outlook on future work in Section 5 we conclude the paper.

2. MIGRATABLE WEB SERVICES

In this section we describe the paradigm of migratable web services. Migratable web services are conventional web services that are capable to change their executing host at runtime without losing the current state. With host we mean the PC providing the web service. The web services of one host are managed and administered by a central software which we call *server*. An example for a server could be the Apache Axis engine [2] or the WebSphere Application Server [10] from IBM. Each server is related to exactly one physical host. A

server and its web services are identified using Uniform Resource Identifiers (URI). Usually, the server is listed in the UDDI-registry with its URIs as binding-point for the web services.
As migratable web services have a client-dependent state we may have several different instances of the same web service. Like objects of the same class in object-oriented programming languages the web service instances share the same code but differ in their current state, e.g. the values of variables. To be more precisely, a client is not bound to a web service directly but to its specific *instance* of the web service.
To become migratable a web service must be serializable, i.e. the current state of an instance can be transformed to a stream which is afterwards sent to another host. The receiving host is deserializing the stream and creates a new instance of the web service with the same state as the original one. In the Java programming language serializability can easily be achieved by implementing the `Serializable` interface [21] in the corresponding source code of a web service.

Server side: Web Service Introspection

Enabling migratable web services requires extensions of the underlying web service runtime environment. Usually a web service is deployed on one server and if different instances of this web service exist they are controlled by the session management of the server. First, the session management gives out identical session ids to the instance and the client and guarantees that further calls from the client are forwarded to the correct instance. In order to support migratable web services we need a way to export (emigrate) and import (immigrate) instances at runtime. A simple but system dependent approach would extend the functionality of the server software directly by changing its source code. This approach is hardly transferable to other server systems because required changes will be different for each system. To be more general, we introduce an approach using a special web service called Migrate-Web Service (Migrate-WS) which is capable of introspecting and migrating other web services hosted on the same server. In order to support migratable web services the server has only to deploy this Migrate-WS. The Migrate-WS itself is not migratable and remains on the server. Our approach is comparable to the *Reflection* functionality [25] in Java.

The general architecture of the Migrate-WS accessing the other web services is illustrated in Figure 2.

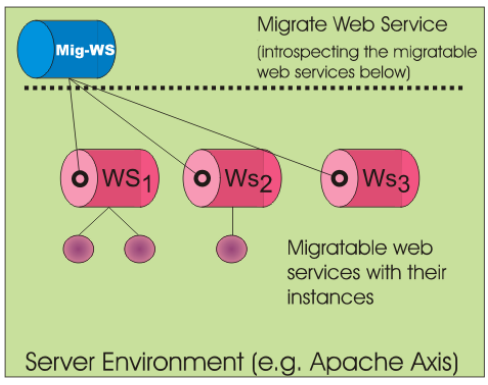

Figure 2. The migrate web service

For each web service the actual instances are displayed: web service WS1 has two instances, WS2 has one, whereas WS3 has no instance in this example. A web service without any instance is a web service which is deployed but not called by any client at the moment. This concept of a dedicated web service that introspects other web services is transferable to most underlying server architectures without significantly changing it.

Details about the Migration Process

The migration can be initiated by any participant in the web service scenario: the web service itself (if it wants to move to the data), the client (who does not want to send its private data) or even a third party (e.g. load balancing surveillant). In each case, the Migrate-WS on the source-side that holds the instance is contacted with an emigrate-call. This call contains the URI of the destination-host and the id of the instance (e.g. the session id). The Migrate-WS serializes the current state of the migratable web service and sends it to the Migrate-WS on the destination-side by calling its immigrate function. If the resources (code, settings, etc.) of the web service are missing on the destination's host they are also transmitted. Both resources and serialized state are sent to the destination within a normal SOAP based call using attachments. The Migrate-WS on the source-side acts as a normal client for the Migrate-WS on the destination-side realizing a migration process with a push strategy. A pull strategy with both Migrate-WS acting contrary is conceivable. On the destination-side the Migrate-WS imports the resources and deserializes the state to a 'living' object that is afterwards announced to the server software. On the source-side the instance is deleted and therefore no longer available.
Beyond this instance-centered migration it is possible to redeploy the whole web service. With redeploy we mean that the web service is undeployed at the source and deployed at the destination. In our judgment, deploying includes the creation of WSDL service descriptions and may include an update of the UDDI-registry because the binding point of the web service has changed. When a web service is deployed on a host, any other client may call it to get an own instance. Migrating without redeploying the service means that only the migrated instance can be used on the host; other clients can not even see that the web service is installed on the host. Undeploying is just the opposite of deploying meaning that the web service as a whole is no longer available at the source-side. All existing clients of this web service are affected.

Whether we can redeploy the whole web service or not relies on its scope. Usually, web services can have three different scopes*: application, request* and s*ession*. The scope of a web service is set when deploying it on the server. Application means that only one instance exists for all clients sharing the same state. Migrating and redeploying this web service means that the one existing instance is removed to another host. In consequence, all clients have to reconnect (transparently by the delegate, see next section) to this host. Migrating an application without redeploying it means that we have two instances which shall share their state. This is only possible using synchronization techniques which makes this scenario much more complex. This problem is not the focus of this paper. If a web service is deployed with the session scope every client gets its own instance. Migrating one of these instances without undeploying it on the source server is the simplest and most common scenario without further problems. The web service can only be undeployed if no other clients are interacting with remaining instances. The request scope indicates that an instance lives only for the time of the current request. Therefore, it makes no sense to migrate the state of an instance. Migrating a web service with the Request scope is synonymous for remote code installation. We summarize the different scopes and the consequences for the migration process in Table 1:

Table 1. Scopes and migration

	Application scope	Session scope	Request scope
Migration and Redeploy	Removal of full application (all clients affected)	Has to be avoided as all other clients with their instances are affected.	Remote code removal (all clients affected)
Migration only	Complicated, as external synchronization is required to keep the states of multiple instances consistent	Normal migration process of one instance (other clients not affected)	Remote code installation (no client affected)

The approach of serialized states requires all participating server environments to support the same programming language, e.g. Java. Of course, this is a restriction that runs counter to the programming language independent model of web services. *Microsoft .NET* [33] provides a shared object model and an intermediate language [6]; this is a promising approach supporting several programming languages like Visual Basic, C, etc. Details about the serialization in .Net and comparisons with Java can be found in [7]. Another approach that is independent of the programming language is to write own implementations for the serializer and deserializer using XML as exchange format. Each language with its own serializer and deserializer can be applied to create the code of a web service.

Client Side: Delegation Model

The main goal of extending a web service system must be the possibility of reusing existing code without changing it. As we showed above, the web service introspection approach of the Migrate-WS approves this for the code on the server side. A consumer of a web service is usually called *client*. The client connects to a web service and calls his methods. Using XML and SOAP messages the communication can be compared with platform and system independent remote procedure calls. The client starts the interaction with the web service by initiating a call to a physical address. This address can be maintained by a registry like the UDDI. In conventional web service architectures the client is now statically fixed to the web service.

In order to support the migration of web services we extended the client side by the use of a so called *Transport Delegate*. The delegate is placed between the client and any remote web service and controls the communication between them. The delegate can be implemented as a web service that runs on the clients host or as an adaptation of the clients implementation of the transport chain as in our implementation. In both cases no adaptation of the client's source code is necessary to operate with migratable web services. The clients call to the remote web service is redirected to the delegate. The delegate extracts the physical address of the call and connects to the relevant web service. If the web service has migrated the delegate has to retrieve its actual location as described later. Responses from the web service to the client are also conducted through the delegate. The connection between the client and its delegate remains static; in this sense, the delegate can be interpreted as a proxy for migratable web services that dynamically change their host. The so far physical client-server connection is now abstracted to a *logical client-server* connection: the client continues to operate with the same logical web service on a different physical host.

The delegate model is also applied by a web service instance itself if it wants to act as a client for another web service. This feature is important if a web service relies on others to provide his service.

In Figure 3 we present the general architecture supporting migratable web services.

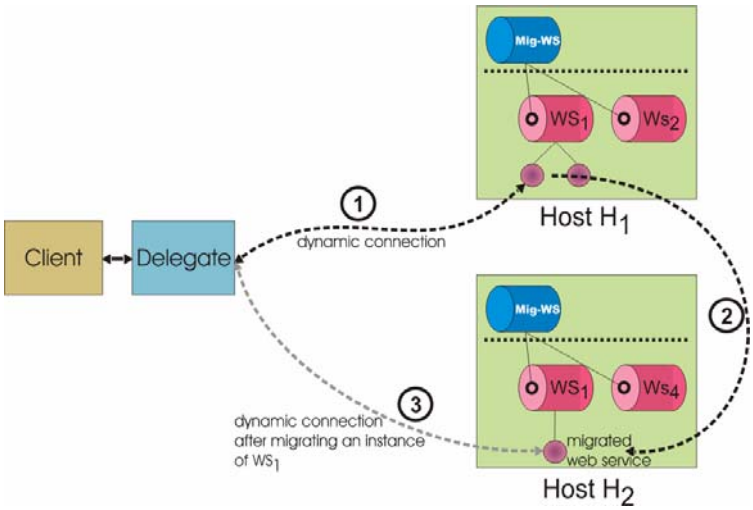

Figure 3. General architecture

The client has established a static connection to its delegate object. First, the delegate has a dynamic connection to an instance of web service *WS*1 on Host H1. After a migration process this instance is hosted on H2. The delegate renews the connection to the relevant instance of *WS*2 by establishing a new dynamic connection to H2. The old connection to H1 is dropped. The client is not affected by the migration process and does not even notice it if we neglect a decelerated response time.

Web Service Instance Discovery

The delegate is responsible for locating the instances of web services for his client after a migration has taken place. When the client connects the web service for the first time the call contains a physical address which is extracted by the delegate and used to establish the connection. After the migration of the instance the delegate communicates with an invalid host because the instance is no longer available here. Therefore, the delegate has to locate the actual host with the instance.

Usually, the UDDI-registry is consulted for locating web services. UDDI registries like [11, 14] are not designed to distinguish between web services (classes) and instances of web service (objects). In addition, the UDDI approach is not constructed for highly dynamic structures that we have in our scenario with migratable web services. The UDDI may be used by the delegate for the first retrieval of a web service before the client-specific instance is created. For further discoveries we need other mechanisms which are described here.

Forwarding of Messages:

As the Migrate-WS on the invalid host knows the current host where he sent the instance he is able to forward the clients requests. This is the easiest approach which requires the delegate to communicate with the Migrate-WS. The delegate does not have to establish a new connection to the current host. But this approach has inherent disadvantages: if we have several migration processes of the same instance we will get a chain of Migrate-WS objects that forward the same request. This will lead to significant degradation of respond times. If one host in the

chain fails, the communication between client and its web service instance is irretrievably broken. The advantage of this approach is that we do not need to relocate service instances. Therefore the delegate in this approach is relatively simple.

Central Notification:

In the second approach the delegate is informed about the new address of the host and establishes a connection to it. Whenever a web service instance is migrated a message to the client or a central instance registry is sent. This approach is illustrated in Figure 3. It results in less active connections than the first approach and is more fail-safe. This approach has the disadvantage that the central registry must still be reachable by the delegate in order to determine the new current host of the instance. When the central registry fails the information about the service instances is lost.

P2P-based Instance Discovery:

The disadvantages of a central registry are solved by a decentralized approach using a peer-to-peer (P2P) grid. In a grid resources are located without the usage of central instances like registries or servers. We interpret a web service instance as a resource that must be announced by the server that hosts it. Every host that receives a web service instance creates an advertisement in order to publish himself as the owner of the instance. The delegate uses a P2P-search engine to retrieve the host that currently hosts the requested web service instance. Each host represents a peer in the P2P-grid.

P2P-based instance discovery is decentralized and allows finding an instance even if previous hosts are not available anymore. An inherent disadvantage of this approach is that the lookup-time is significantly longer than in the other two approaches. The architecture using a P2P-based delegate is illustrated in Figure 4.

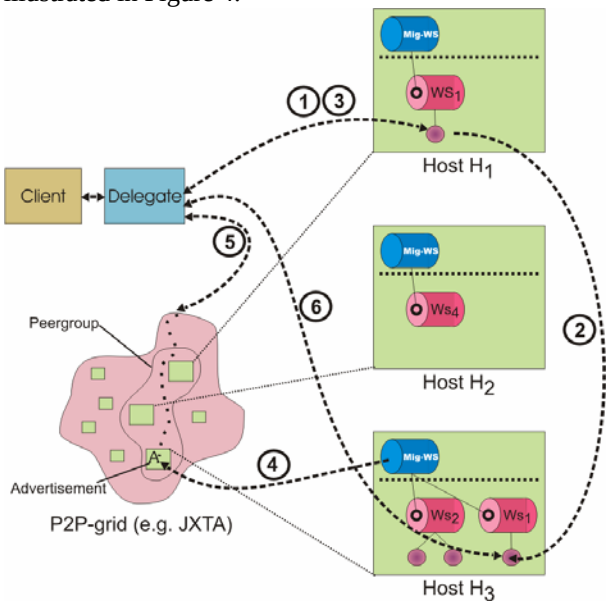

Figure 4. Locating instances using a P2P-grid

The Hosts H_1 to H_3 are represented as peers in the global grid. In step 1 the delegate communicates with a web service on host H_1. After a migration process (2) the delegates request fails (3). At the same time H_3 sends an advertisement to the grid in order to show that he is the new host (4). The delegate finds the advertisement (5), extracts the URI, and connects to H_3 (6).

3. IMPLEMENTATION AND EXPERIMENTS

We implemented the Migrate-WS web service and the delegate on top of the *Apache Axis* [2] engine. Axis is an open source implementation of the web service standard SOAP under license of the *Apache Software Foundation* [1]. Axis uses *Tomcat* [28] as container and supports SOAP1.1 [30] as lightweight protocol for information exchange and WSDL1.1 [31] to describe interfaces of web services. WSDL documents are created automatically. We have chosen Axis because it is open source with manageable complexity.

Implementation of the Migrate-WS and the Delegate.

The Migrate-WS is implemented in Java as a standard (non-migratable) web service providing the two main methods `emigrate` and `immigrate` to relocate the instances of migratable web service. Some auxiliary functions are used to retrieve the list of deployed web services by calling a reflection method provided by the Axis-engine. Analogously to the *MessageContext* in Axis we defined a *MigrationContext* class containing all information required for the migration process:

- the URI of the destination Migrate-WS,
- the qualified service name and the clients id,
- redeployment information.

The MigrationContext is a Java Bean that is serialized into a SOAP-message using the standard Bean serializer from Axis and passed to the Migrate-WS on the source-side which starts the migration process. All communications of our implementation relies on the HTTP protocol, although Axis supports other protocols.

The delegate is implemented on the client side as part of the global handler-chain of Axis. Each call of the client is handed through this chain, thus the delegate can redirect calls to the current host without any interaction of the clients code. Our implementation is an extension of Axis without requiring change in the Axis code. Existing web service components can easily become migratable: there are no changes required on the client's implementation and the web service only needs to implement the `Serializable` interface from the Java programming language.

We implemented a graphical user interface (GUI) that informs the user about deployed web services on a set of servers. The GUI acts as a client for the Migrate-WS and calls its introspecting. Figure 5 shows a screenshot of the user interface.

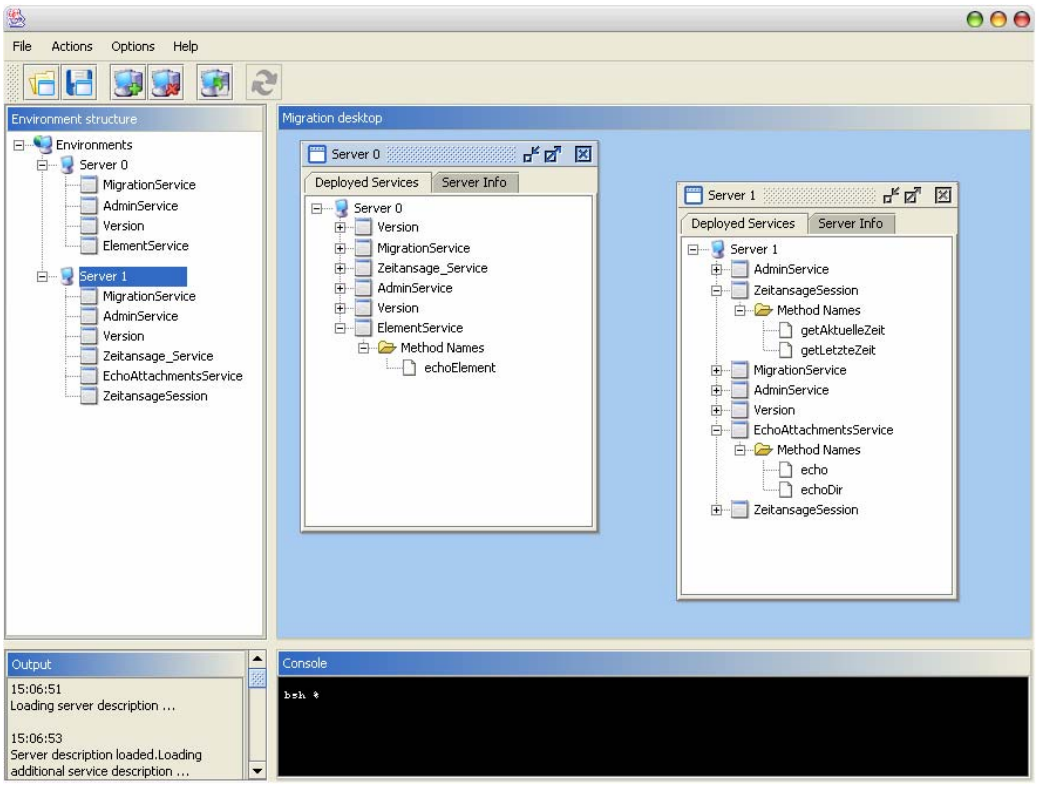

Figure 5. Screenshot

JXTA based P2P Instance Search Engine

Discovering instances of web services is a task which belongs to the delegate. We realized the notification approach as described in Section 2.4. In addition, for a more failsafe operating when the source-host is already down we explored P2P-based discovering using Suns *JXTA* [24, 34] approach. The JXTA technology is a set of open, generalized peer-to-peer protocols that allows any connected device to communicate and collaborate. JXTA is an open source effort. In our implementation each server on each host is represented by one peer in the P2P grid. All peers are summarized in a dedicated peer-group in order to separate them from other peers that do not belong to the migratable web service context. Whenever a Migrate-WS immigrates a new service instance it creates an advertisement that is published in the grid. The advertisement contains information like the hosts URI and the name and the id of the instance. An example for a JXTA advertisement is displayed in Listing 1.

The tag `GID` states the Peergroup of the Host. `MSID` is the JXTA-generated id of the web service interpreted as a resource (as we use a *Module Specification Advertisement*). The `Name` tag identifies the Web Service in general whereas `InstanceID` is the ID of the instance which stays the same for the lifetime of the instance. This unique id is generated automatically by the Migrate-WS which initiates the first migration.

```
<?xml version="1.0"?>
<!DOCTYPE jxta:PGA>
<jxta:PGA xmlns:jxta="http://jxta.org">
<GID>urn:jxta:jxta-NetGroup-MigWs</GID>
<MSID>urn:jxta:uuid-A0783B698094493295E...</MSID>
<Name> JXTASPEC:SCH:ImgManipWS </Name>
<InstanceID>1692549811281</InstanceID>
<Location>169.254.96.9</Location>
<Desc>Web Service for Image Manipulation</Desc>
</jxta:PGA>
```

Listing 1: Advertisement for a web service instance

The delegate queries the P2P grid using the *Peer-Discovering-Protocol* of JXTA when he recognizes that an instance is no longer available at the previous host. The query is initiated by a simple method in Java and executed by the underlying JXTA framework which returns a list of potential hosts. Using the extension *JXTA Search* [27] we want to reduce the query response time. JXTA Search is tailored for environments where content is rapidly changing and is spread out across many different providers.

Testing Scenario and Performance Measurements

In order to prove the efficiency and the performance of migratable web services we set up a scenario that pays attention to data privacy (example 1 in the introduction) and client-centric performance tuning (example 2): A client uses a financial web service managing a portfolio of personal stocks. The type and amount of stocks and their summarized value differ for each client; therefore they build an individual state of client's web service instance. In order to support multiple instances we deployed this web service with the session scope. Figure 6 illustrates the test scenario.

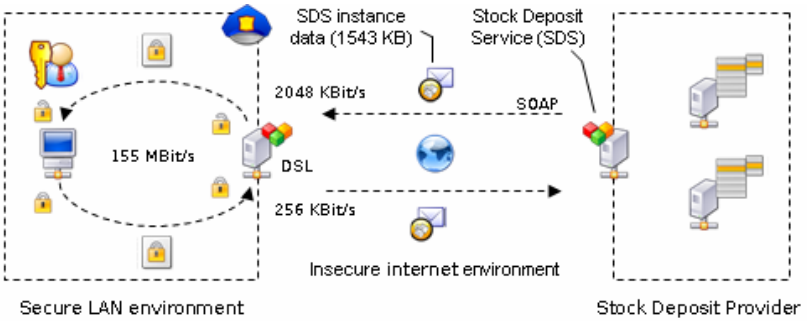

Figure 6. Test scenario

On the left side, the client acts in a secure environment – his local network that might be secured by firewalls, sandbox mechanisms, etc. The communication with the web service provider on the right side is performed using a DSL connection. The connection is asymmetric providing a 2048 KBit/s downstream but only a 256 KBit/s upstream from the client's side. Therefore, it makes a difference whether data is transmitted from or to the service instance.

In this test scenario we created a stock portfolio with a typical size of 1.5 MByte whose data is sent to the service instance and afterwards back to the client.

In a first run we evaluated this scenario with a standard web service that is not able to be migrated. We create 10 calls of a web service method that requires the portfolio data as input. This data is sent 10 times from the client to the service. Each method call performs some processing on the portfolio data and sends it back.

In the second run the web service instance is migrated into the clients secure environment before processing the 10 method calls. As a consequence there is never an upload of the portfolio data to the service provider on the right side of figure 6. All method calls can be performed within the local network of the client.

In Table 2 we compare the measured times of both runs. The migration includes the serialization, transmission and deserialization of the instance. One can see that migratable web services increase an average method call by more than a factor of 7. In addition, the client may use an unknown server because all personal data is kept in the secure environment of the client.

Table 2. Comparing conventional web services and migratable web services

	Run 1: Conventional ws	Test 2: Migratable ws
Migration efforts	-	5 sec
Request (client to server)	76 sec	2 sec
Method execution	4 sec	4 sec
Response (server to client)	14 sec	2 sec
Total execution time (per call)	94 sec	13 sec

4. RELATED WORK

In this chapter we compare our approach of migratable web services with the state-of-the-art in conventional web service environments, mobile agent based systems, grid computing and P2P-networks.

Web Service Environments: Todays server software for hosting web services like *Apache Axis* [2], IBMs *WebSphere Application Server* [10] or the Microsoft .NET Framework [33] do not offer migratable web services directly as it is not a W3C requirement for web services. The idea of stateful web services and how to model them is introduced in [9] although the migratability of the web services is not treated. The intermediate language of .NET and its Common Language Runtime (CLR) is a promising approach relieving the efforts spent to support different programming languages in a migratable web service compound. An evaluation how to support mobile code in .NET can be found in [18]; although the authors do not deal with web services in general. An architecture providing an agent system based on .Net is presented in [20], but this approach requires significant .NET relevant changes and is

not generally transferable to other web server environments. The approach of ObjectGlobe [3] proposes a system that is capable to send services to remote hosts and deploys them there. But as there is no discrimination between web services and instances the current state of a web service is ignored. Therefore this approach is restricted to remote code installation.

Mobile Agent Systems: Mobile Agents can be defined as follows [29]: *A mobile agent is a program that can migrate from host to host in a network of heterogeneous computer systems and fulfil a task specified by its owner. It works autonomously and communicates with other agents and host systems. During the self-initiated migration, the agent carries all its code and the complete execution state with it. Mobile agent systems build the environment in which mobile agents can exist.*
First, one can say that mobile agent systems offer the migration possibilities that are required for migratable web services. But agents have a different perspective concentrating on the autonomous behaviour and intelligence. Independence of platforms or programming languages, universal description and discovering of services is ,if at all, a minor goal. Even if an agent system is service-oriented, i.e. it tries to act as a service provider (e.g. [19]) standard web service related languages like WSDL or SOAP are usually not supported. Because the migration can be initiated by the web service's instance itself our approach provides autonomous behavior; in this case the instance can be regarded as a mobile agent.

Peer-to-Peer Networks and Grid Computing: Common P2P-applications like the file sharing tools *Kazaa*, *Gnutella* or *eDonkey* are used for sharing resources – in most cases the resources are restricted to files like movies which can be downloaded by participants of the network. Even if we interpret the download process as a migration act these P2P-systems cannot be regarded as hosts for web services as they do not support any execution of code. Grids used for distributed computing like the popular *SETI@home* project seeking for extraterrestrial intelligence do support the migration of code but they are usually restricted to one application and do not offer web service characteristics as the description of interfaces for instance. A web service related approach supporting the composition of loosely coupled services in a grid is presented in [12]. The possibility of those web services to migrate between hosts in the grid by moving the current state is not mentioned. The same is true for the open grid services architecture *Globus* presented in [4, 5]. This system allows the location transparent usage of web service instances distributed over the grid. In addition, Globus supports the description of services using web service standards like WSDL. Because Globus is a new implementation the architecture may not be transferable to existing web service environments. Like in our approach the author think of integrating a JXTA based discovering tool.

We call our web services *'migratable'* instead of *'mobile'* although the mobility of the web service's code suggests the latter. The term 'mobile web services' is often used for a system supporting web services on *mobile devices* like cell phones or PDAs mostly without moving any code.

5. CONCLUDING REMARKS

In this paper, we proposed the – to our knowledge new – paradigm of fully migratable web services using P2P for discovering instances. With our approach web services can relocate their executing host without loosing their current state. A client's connection to the web service is not affected. Existing code on client-side and server-side can be reused without changes. The data exchange between client and server still relies on web service specific protocols like SOAP or WSDL. The so far physical Client-Server connection is now abstracted to a *logical Client-Server* connection without loosing the general usability. We provided and implemented a general approach to enable conventional server software like *Apache Axis* to support migratable web services. The implementation includes a P2P-grid based on Suns *JXTA* for locating migrated instances of web services without using a central registry. With some measurements within a test scenario we proved that migratable web services may increase the total performance of a web service architecture and increases the consumers' satisfaction due to decreased response times and network requirements.

Future work is twofold: first we want to specify the migration characteristics of a web service in general by extending the web services deployment language *WSDD* and *WSDL*. In order to perform more measurements we plan to implement a load balancing tool that automatically migrates highly requested web services on a multi-PC cluster when the performance limit of one PC in the cluster is reached.

ACKNOWLEDGMENTS

The authors thank Ivo Iken and Frank Müller for implementing wide parts of the Apache Axis extensions, the graphical user interface and the JXTA search engine for web service instances within the scope of a student research project.

REFERENCES

[1] Apache. The Apache Software Foundation. URL: http://www.apache.org/foundation.

[2] Apache Web Services Project. Axis. URL: http://ws.apache.org/axis/.

[3] R. Braumandl, M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam, K. Stocker: ObjectGlobe: Ubiquitous query processing on the Internet. VLDB Journal: Very Large Data Bases, 10(1):48–71, 2001.

[4] I. Foster, C. Kesselman, J. Nick, and S. Tuecke: The physiology of the grid: An open grid services architecture for distributed systems integration, Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[5] I. Foster, C. Kesselman, and S. Tuecke: The anatomy of the Grid: Enabling scalable virtual organization. The International Journal of High Performance Computing Applications, 15(3):200–222, Fall 2001.

[6] D. R. Hanson. lcc.net: Targeting the .NET Common Intermediate Language from Standard C. Software: Practice and Experience, 34:265 – 286, 2004.

[7] M. Hericko, M. B. Juric, I. Rozman, S. Beloglavec, A. Zivkovic: Object serialization analysis and comparison in java and .net. ACM SIGPLAN Notices, 38:44 – 54, 2003.

[9] Ian Foster, Jeffrey Frey, Steve Graham, Steve Tuecke, Karl Czajkowski, Don Ferguson, Frank Leymann, Martin Nally, Igor Sedukhin, David Snelling, Tony Storey, William Vambenepe, Sanjiva Weerawarana: Modeling Stateful Resources with Web Services. URL: http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.html, 2004.

[10] IBM Corp. Websphere platform. URL: http://www.ibm.com/websphere.

[11] IBM Software. Web services by IBM: UDDI. URL: http://www-306.ibm.com/software/solutions/webservices/uddi.

[12] F. Leymann and K. Güntzel: The business grid: Providing transactional business processes via grid services. In M. E. Orlowska, S. Weerawarana, M. P. Papazoglou, and J. Yang, editors, Proceedings of the First International Conference of the Service-Oriented Computing (ICSOC 2003), volume 2910 of Lecture Notes in Computer Science, pages 256–270, Trento, Italy, December 15-18, 2003. Springer.

[14] Microsoft. UDDI Business Registry (UBR) node). URL: http://uddi.microsoft.com/default.aspx.

[15] OASIS. UDDI: Advancing Web Services Discovering Standard. URL: http://www.uddi.org.

[16] OASIS. Web Services Security v1.0 (WS-Security 2004). URL: http://www.oasis-open.org.

[18] G. P. Picco and M. Delamaro: Mobile code in .net: A porting experience. In N. ed., editor, In Proceedings of the 6th International Conference on Mobile Agents (MA 2002), volume 2355 of Lecture Notes on Computer Science, pages 16–31, Barcelona, Spain, 2002. Springer.

[19] P.-A. Queloz,A. Villazon: Composition of services with mobile code. In Proceedings of the First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99), Palm Springs, CA, USA, 1999.

[20] A. R´equil´e-Romanczuk, C. Mingins, B. Yap, O. Constant. Leopard: a .net based agent architecture. In Proceedings of the 2. International Joint Conference on Autonomous Agents & Multiagent Systems (AAMAS), July 14-18, pages 1108–1109, Melbourne, Victoria, Australia, 2003. ACM.

[21] Sun Microsystems. Java Object Serialization. URL: http://java.sun.com/j2se/1.3/docs/guide/serialization.

[24] Sun Microsystems. The Project JXTA. URL: http://www.jxta.org.

[25] Sun Microsystems. Using Java Reflection. URL: http://java.sun.com/developer/technicalArticles/ALT/Reflection.

[26] Sun Microsystems. White Paper: Secure Computing with Java: Now and the Future. URL: http://java.sun.com/ security/javaone97-whitepaper.html.

[27] Sun Microsystems - Project JXTA. JXTA Search. URL: http://search.jxta.org.

[28] The Apache Jakarta Project. Apache Tomcat. URL: http://jakarta.apache.org/tomcat/.

[29] The Software Engineering Group of the Computer Science Department of the Friedrich Schiller University Jena , Germany. The Mobile Agent System Tracy. URL: http://tracy.informatik.uni-jena.de.

[30] The World Wide Web Consortium (W3C). Simple Object Access Protocol (SOAP) 1.1). URL: http:// www.w3.org/TR/soap.

[31] The World Wide Web Consortium (W3C). Web Services Description Language (WSDL) 1.1). URL: http://www.w3.org/TR/wsdl.

[32] R. K. Thomas, R. Sandhu: Models, protocols, and architectures for secure pervasive computing: Challenges and research directions. In In Proceedings of the 2. IEEE Annual Conference on Pervasive Computing and Communications Workshops,March 14 - 17, pages 164–170, Orlando, Florida, USA, 2004.

[33] T. Thuan and L. Hoang. .Net Framework Essentials. O'Reilly & Associates, 2002.

[34] B. J. Wilson. JXTA. New Riders Publishing, 2002.