An Approach for Fine-Grained Web Service Performance Monitoring

Jan Schaefer

Fachhochschule Wiesbaden - University of Applied Sciences
Distributed Systems Lab
Kurt-Schumacher-Ring 18, D-65197 Wiesbaden, Germany
jan.schaefer@informatik.fh-wiesbaden.de

Abstract. Especially for the creation of *Service-Oriented Architectures* (SOA), Web service technologies are often *the* technology of choice. In this context, solutions for the management of Web services are becoming more and more important. This paper describes an approach to performance monitoring of Web services, which is based on the *Application Response Measurement* (ARM) standard. This approach enables generic (application source code-independent) and customizable instrumentation of synchronous, asynchronous and one-way Web service messages by attaching meta-data to messages.

1 Motivation

Integrating a company's existing software assets into a *Service-Oriented Architecture* (SOA) is gaining enormous popularity. This is caused by the promise of service unification and increased software reusability on the one hand and the evolution of Web service technologies, which are the most common SOA building elements, on the other hand. Companies have discovered the possibility of modernizing their legacy systems without necessarily having to modify their existing applications. To accomplish this, Web services are often created as wrappers for existing applications. Companies can benefit from a SOA in many ways. For example, they can use orchestration languages like the *Business Process Execution Language* (BPEL) to create new services by combining their existing (Web) services (called *service composition*). Although Web service technologies are a relatively easy way of integrating existing and newly developed applications, their interfaces also add another layer of complexity to applications. This adds to the importance of being able to monitor distributed systems in a homogeneous way. On one hand, developers might be interested in testing the effect of their changes on throughput and performance (e.g. response time) of system components. This includes, for example, the total processing times of single requests, the processing times in client and Web service or the transport times. On the other hand, administrators might primarily be interested in generic monitoring or tracking of failed or delayed transactions or performance bottlenecks. They could also use the detailed runtime information (e.g. execution times, states,

F. Eliassen and A. Montresor (Eds.): DAIS 2006, LNCS 4025, pp. 169–180, 2006.

values) to check *Service Level Agreements* (SLA). In recent years, birth has been given to several Web services-supporting platforms (e.g. Apache Axis [1], IONA Artix [2] and Microsoft .NET [3]). The vendors all claim full interoperability with each other's platforms, which is a major argument for using Web service technologies in the first place.

Web services management is still a relatively young discipline. Some *Enterprise Management Systems* (EMS) support managing Web services-based software and hardware (e.g. IBM TCAM [4] and CA Unicenter WSDM [5]). However, management capabilities for single and composed Web service transactions are still rare. Existing solutions for this are custom-tailored to specific Web services products and thus not easily deployed in heterogeneous environments. None of the existing Web services management specifications covers the monitoring or analysis of single Web services transactions. Instead, they confine themselves to monitoring deployed Web service applications and hardware devices (called *resource management*). The two specifications in this area are *Web Services for Management* (WS-Management [6]) and *Web Services Distributed Management* (WSDM [7]) with partially overlapping aims. This situation was additional motivation for the work presented in this paper (complete report in [8]).

The approach presented in this paper offers an instrumentation solution for Web services based on the *Application Response Measurement* (ARM) standard. In order to keep the approach as generic as possible, it does not require specific management agents or modification of the application to be instrumented. It relies on standardized specifications that multiple vendors incorporated into their products. It focuses on the timing of synchronous and asynchronous Web service invocations to gather performance-related measurement data. In this context, it has to be considered that multi-threaded processing and asynchronous messaging introduce additional requirements (e.g. for request and response message matching) in comparison to classical RPC-style interaction.

2 Application Response Measurement

The ARM standard, whose development is overseen by The Open Group, provides an API for instrumenting applications at the source code level [9]. The API supports execution time measurements of work units termed *ARM transactions* within distributed applications. ARM allows correlating nested measurements, even across host boundaries. For this purpose, the standard defines *ARM correlators*, which are unique tokens assigned to each ARM transaction. Correlators can be supplied when creating a nested transaction for relating this to the enclosing transaction. Passing correlators between application components, which might prove difficult especially in distributed systems, is the task of the application developer. ARM allows for the integration of applications directly with enterprise management systems. This creates a comprehensive end-to-end management capability, including the measurement of application performance, availability, usage and end-to-end transaction response time. To effect this integration, developers have to add ARM calls to their application code, which

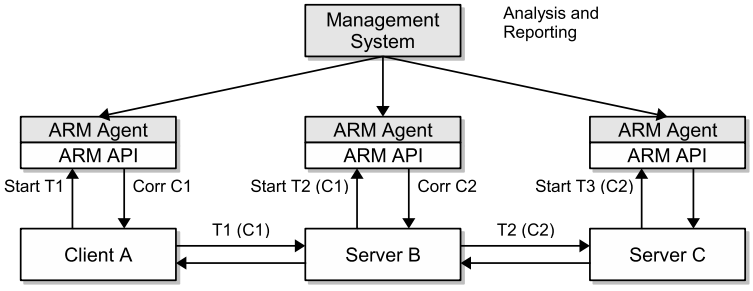

Fig. 1. Transaction Correlation using ARM API

are processed by an ARM agent during application execution. The process of finding relevant measurement points and inserting measurement code is called *instrumentation*.

The management agent collects status, response time and – optionally – additional measurement quantities associated with the transaction (see figure 1). Together with the agent, the instrumented application may also provide information to correlate parent and child transactions. For example, a transaction that is invoked on a client may drive transactions on an application server, which in turn drives other transactions on other application and/or database servers. This allows the construction of a calling hierarchy that illustrates which transactions are *nested* into or dependent on others in subsequent transactions. The example in figure 1 leads to the following ARM transaction hierarchy: server C uses the correlator received from server B, which uses the correlator received from client A. Thus, the ARM transaction C depends on B and B depends on A.

ARM measurement results and correlations have to be evaluated by ARM implementations (agents), which are available, for instance, from BMC, CA, HP, IBM (Tivoli) and tang-IT. Their implementations and analysis tools are often integrated with their respective management solutions and quite different from each other. However, The Open Group also provides a free SDK, which contains implementations of the standardized interfaces and can be used for testing and validating instrumented applications.

The ARM standard is developed by members of The Open Group, namely IBM, HP and tang-IT. With the release of ARM 4.0, the available C and Java bindings provide equivalent functionality for the first time. The approach presented here uses the Java binding of ARM 4.0 [10], which contains new features such as asynchronous reporting of transaction information.

3 Web Services and ARM

3.1 Message Exchange Patterns

The *Web Services Description Language* (WSDL [11]) defines four transmission primitives: one-way (client to service), request-response (client to service and

back), solicit-response (service to client and back) and notification (service to client). This paper concentrates on the following *Message Exchange Patterns* (MEP), because they describe the set of exchanged messages for the primitives (in-out and in-only with changing direction):

- *Synchronous request-response*: the client sends a request to the service and blocks until it receives the response from the service.
- *Asynchronous request-response*: the client sends a request to the service and continues processing. The client either has to check for response arrival (e.g. by polling), or it has to enable the service to invoke it (e.g. by offering a callback method).
- *One-way*: the client sends a request to the service and continues processing without blocking or expecting a response.

3.2 SOAP Message Handlers and Contexts

SOAP [12] *Message Handlers* and *Message Contexts* are both defined in the *Java API for XML-based RPC* (JAX-RPC [13]) specification, which supports building Web services that use *Remote Procedure Calls* (RPC) and XML. The JAX-RPC API hides the complexity of SOAP messages from the developer, and the runtime system converts the API calls to and from SOAP messages. JAX-RPC supports stateless message handlers (also known as *interceptors*), which allow the modification of messages before and after they have been dispatched to a service or client implementation (e.g. to add security or management information). To use handlers, no application level code has to be modified. Instead, handlers are added through deployment configuration. There are two types of message handlers: client- and server-side handlers. They are invoked depending on their associated Web service's or client's role in the message exchange (see figure 2), and their order is determined by their deployment configuration. Message handlers implement the *Chain of Responsibility* design pattern, which means that a message is processed by all handlers (in a handler chain), before it is dispatched to the targeted service. JAX-RPC also supports one-way messaging in addition to the request-response messaging style normally done with RPC. JAX-RPC is supported by most Web service platforms.

Message contexts are used to store meta-data about messages (e.g. security or management information) or to exchange state information between application

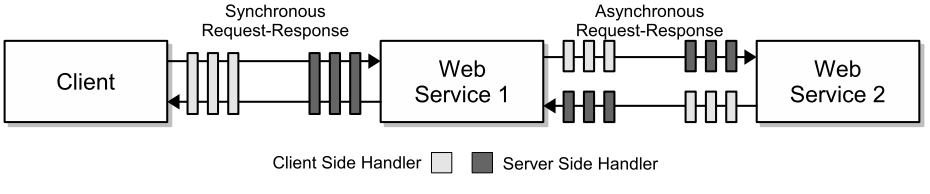

Fig. 2. Message Handler Chains

level and message handling chain. Generally, this meta-data does not cross host boundaries, but some platforms allow attaching it to messages. This enables a meta-data exchange between clients and services.

3.3 Instrumentation Challenges

This paper presents a solution for generic ARM-based instrumentation of Web service platforms. For this, the subsequently introduced problems had to be solved.

1. *Isolation of instrumentation code*: Often, instrumentation is a static process. Instrumentation calls are inserted into source code, which means that every application has to be instrumented explicitly (hard coded). If the application code is modified, the instrumentation code may have to be modified as well. This process is error-prone and slows down development speed. In addition, developers must know the ARM API. For Web services, this means that every Web service – depending on the scope of instrumentation – would have to be instrumented manually by experienced ARM users.
2. *Support for different message exchange patterns*: In an asynchronous message exchange (messaging scenario), both the request and response messages are defined as a one-way message in the WSDL contract. Thus, they are only semantically related, and only the application logic "knows" the meaning of received response messages. If a client receives a related response directly from a previously invoked service, the response is received *directly*. If a client receives a response from a different service, the response is received *indirectly*. As a result, a mechanism for relating request and response messages has to be created, which allows starting and stopping ARM transactions (measurements) correctly.
3. *Transport of ARM data and correlators*: Another problem arises when thinking about how ARM data has to be exchanged between services. Of course, it must be meta-data rather than an invocation parameter. To correlate ARM transactions, correlators must be propagated, because whenever a client or service receives a message (even if it is a response message!), the correlator might be required for starting a new (dependent) ARM transaction.

4 Architecture

4.1 Design Decisions

This paper presents a solution for generic rather than manual instrumentation as described in section 3.3.1. Therefore, the JAX-RPC message handler and message context mechanisms were selected as hooks for the ARM-based instrumentation. More specifically, message handlers encapsulating the ARM-related code were created for intercepting and augmenting messages, and message contexts were defined for storing the ARM-related meta-data required during the instrumentation process.

For in-out MEPs, the ARM handlers are used to measure the total processing time of business transactions, from the moment a request is sent, until its related response is received (a client-side measurement). It is also possible to use ARM handlers for measuring the service's response time per invocation only (a server-side measurement). By using both client- and server-side ARM handlers, it is possible to calculate message transfer times as well. The instrumentation by configuration allows inexperienced ARM users to instrument services; no knowledge of the ARM API is required and existing service implementations do not have to be modified. In addition, experienced ARM users can manually instrument services to gain more fine-grained performance data. Using configurable ARM handlers for instrumentation is defined as *system-level instrumentation*, manually instrumenting services is defined as *user-level instrumentation*. The developed instrumentation solution presented here allows using both system- and user-level instrumentation simultaneously. In addition, nested ARM transactions can use ARM correlators created by the enclosing ARM transaction for correlation: correlators are propagated to succeeding services, even across host boundaries (see section 3.3.3). ARM handlers and service implementations can access propagated correlators via the ARM context.

Even in a synchronous message exchange, handlers process messages asynchronously. Otherwise, a synchronous (blocking) call would prevent the Web service from processing multiple requests in parallel. Thus, the instrumentation model does not differentiate between synchronous, asynchronous and one-way communication. Furthermore, request and response are not even processed by the same message handler in asynchronous exchanges: the request is processed by a client-side handler, the response is processed by a server-side handler (see figure 2). Because of this, matching of request and response messages is provided by inserting unique ARM transaction IDs into the ARM message context of related messages, which enables ARM handlers to recognize relationships between messages (see section 3.3.2). Transaction IDs are also used for identifying ARM transactions: if an ARM handler is configured to stop a running ARM transaction, it uses the transaction ID inside the ARM context for the look-up. If the ARM context should be removed or missing, no messages can be matched and no ARM transactions can be stopped anymore.

Depending on its configuration, an ARM handler might have to store a received ARM context, which has to be returned in the response message, and a reference to a started ARM transaction. This information is stored in the ARM registry. If required (respectively configured), the registry information is used to restore a parent context and to stop an ARM transaction. This is required, because the processing of related request and response messages in message handlers can be interrupted due to multiple concurrent threads. However, each Web service (respectively its ARM handlers) only has to store the first received ARM context in each business transaction. All sub-requests that are created by this service have to use this context. Of course, ARM transaction references have to be stored for every started transaction.

Once the Web service finishes processing the business transaction (and a service's ARM handler is invoked for the last time), the parent ARM context is returned to the client inside the response message. Should the current Web service be only an intermediary in the business transaction, the parent ARM context is propagated to the next service. In in-only MEPs, the context of instrumented messages contains a time stamp, which denotes the start time of the associated ARM transaction, and an ARM correlator. One-way messages do not contain ARM transaction IDs in their message context, because they will not return to the invoking client. Thus, no request-response matching is required.

4.2 Overall Structure

Figure 3 shows the component interaction of the generic instrumentation model, which solves the problems presented in section 3.3.

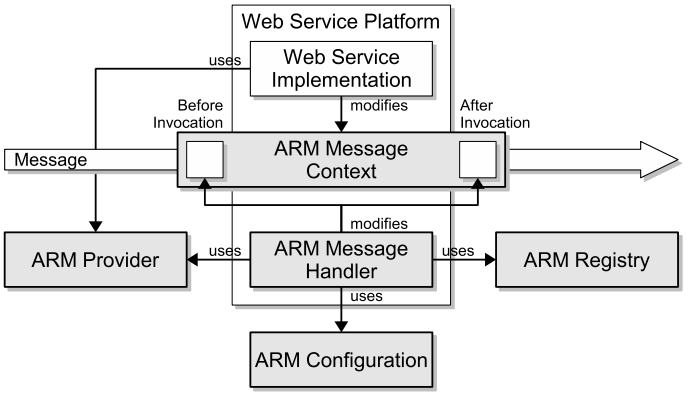

Fig. 3. Architectural Components

The *ARM Message Handler* component is responsible for intercepting and processing messages, before and after the invocation of the handler's associated Web service. It represents the main component of the instrumentation solution and provides client- and server-side message handlers that can be configured to instrument Web service message exchanges. The purpose of the handlers is to provide ARM operations that can be specified using the ARM configuration, based on handler type (client- or server-side), Web service name and operation name (see figure 4).

The *ARM Message Context* component represents a container for transferring ARM information that is exchanged between Web services in instrumented transactions. The context contains one section for use by ARM message handlers (system-level) and one for use by Web service implementations (user-level). Service code may read the system-level section of the context (e.g. for using the ARM correlator within as a parent correlator in user-level ARM transactions),

but it must write to the user-level section only. For supporting one-way message exchanges and ARM transaction reporting, the context also contains a field that can hold a time-stamp (the start time of a reported transaction). Finally, the context contains a context ID. The context referenced by this ID is used by sub-transactions for retrieving the parent ARM context. Each of the available elements of the context is optional, so that they are only put on the wire if required.

The *ARM Provider* component encapsulates access to the ARM API, which for this approach was the official ARM 4.0 SDK (available from [9]). The provider also executes a base ARM application, which can be used for user- and system-level instrumentation. This avoids the overhead of creating an additional ARM environment, if there is no specific need for it.

The *ARM Configuration* component is queried for the action to be executed, whenever an ARM handler intercepts a message. A handler might have to start a new ARM transaction, stop a previously started ARM transaction or stay idle (context forwarding only). This allows users to keep the impact of the instrumentation on the performance of the rest of the system at a minimum. However, a handler alone cannot decide how to handle a received ARM context: it cannot be specified programmatically, which incoming and outgoing messages represent the start and end of a business transaction; the user has to define the appropriate action in the configuration (figure 6 shows a configuration example). When a service receives a request, it must save the (parent) ARM context even if no ARM action is executed immediately, because sub-transactions might require the context for starting new ARM transactions; using the correlator inside the parent context is mandatory for correct ARM transaction correlation. In the end, the handler processing the response message that is returned to the invoking client restores (and unregisters) the parent context.

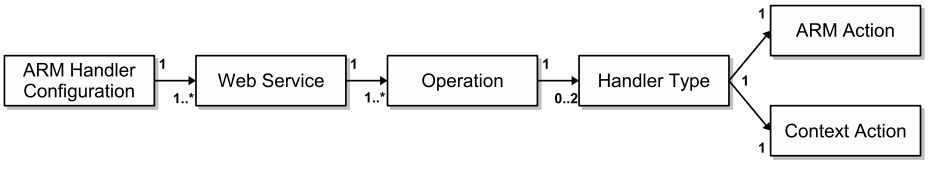

Fig. 4. ARM Configuration Structure

The configuration contains the following handler configuration for every operation a Web service provides or interacts with (see figure 4):

- The *ARM Action* attribute controls which instrumentation-related action an ARM handler must take: *idle* (handler stays passive and only forwards an existing ARM context), *start* (an ARM transaction) or *stop* (an ARM transaction).
- The *Context Action* attribute controls how an ARM handler treats the current ARM context: *idle* (use current context), *use* (the parent context), *save*

(register current context in ARM registry) or *restore* (the parent context and destroy the local copy in the ARM registry).

The *ARM Registry* component stores copies of ARM contexts and ARM transactions and serves as shared memory for all associated ARM handlers. It handles three types of records: One that contains active ARM transactions with associated ARM contexts, one that contains ARM contexts only and a third that contains currently inactive ARM transactions. The registry checks the stored records cyclically for timed-out ARM transactions and ARM contexts. For timeout checking, the stored items contain a time-stamp that is updated when a record is created or used. If an ARM transaction exceeds the maximum processing time (which is configurable), it is aborted and returned into the pool for inactive ARM transactions.

The ability to store ARM contexts is required to be able to undo changes to message contexts caused by new ARM transactions and is only required for instrumentation in ARM handlers, not for manual instrumentation. The active ARM transactions stored in the ARM registry are required for stopping them, once they are finished. The registry uses unique identifiers as keys for retrieving stored records. These keys are put into the ARM context as references to the records.

5 Prototypical Implementation

Initially, the instrumentation prototype was intended to be used to instrument Web services using both Apache Axis 1.2 and IONA Artix 3.0 as platform. However, it turned out that Axis lacks support for asynchronous WSDL operations. Thus, the prototype currently supports Artix only, but it could easily be ported to different Web service platforms implementing JAX-RPC, respectively platforms incorporating the message handler concept.

Artix is IONA's commercial Web services-based solution for *Enterprise Application Integration* (EAI). Artix supports multiple transports and message formats natively. It connects applications at the middleware transport level and translates messages only once using direct on-the-wire transformation instead of a canonical format. An open source offspring of Artix – called *Celtix* – is hosted by ObjectWeb [14].

Artix supplies the generic classes (*GenericHandler* and *GenericHandlerFactory*) implementing the JAX-RPC interfaces (*Handler* and *HandlerFactory*), which can be extended by developers to implement custom handlers. The handler implementation has to be wrapped in a plug-in, which can then be loaded by Artix clients and servers. The handler mechanism allows intercepting and modifying messages at four points of a message exchange. Both request and reply message can be handled at the client request-level, the client message-level, the server message-level, and the server request-level. Handlers at the request-level have access to the application's message context and the message's SOAP header respectively it's security properties. Handlers at the message-level have access to

the raw message stream that is being written out on the wire and the application's message context only. For Artix, the instrumentation model presented in section 4 was implemented as a request-level handler, which can be configured using an XML configuration file.

The prototype was evaluated using a lab-level travel agency scenario featuring five interacting Web services: *Customer*, *Travel Agency*, *Airline*, *Car Rental* and *Hotel* (see figure 5). In this scenario, a customer sends booking orders to the travel agency, which then books flight, car and hotel room for him. Once the agency received all required information, it returns a booking confirmation to the customer. The services basically react on received requests by sending appropriate responses (containing dummy data), which means that they do not execute complex algorithms. The purpose of this scenario is to prove the applicability of the approach for asynchronous communication. Thus, all communication in this scenario takes places asynchronously, and all Web service operations are defined as WSDL one-way operations. The performance measurements were executed using one Artix client (providing the Customer callback service) and one Artix server (hosting the remaining Web services). Both applications were executed on different hosts.

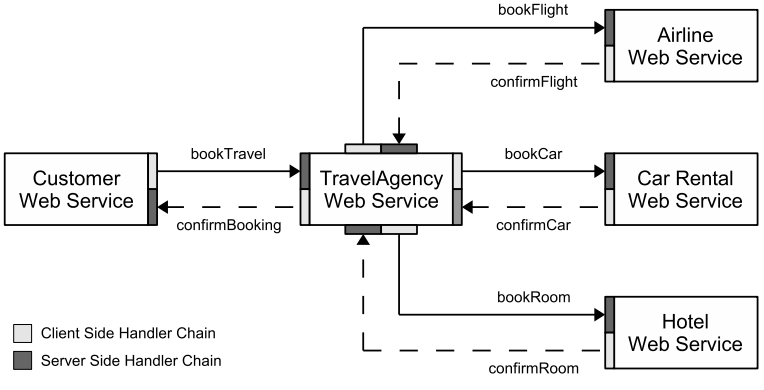

Fig. 5. Travel Agency Scenario (Messaging)

Figure 6 presents an extract of the travel agency scenario shown in figure 5 and the associated configuration. In this example, the communication between the travel agency and the airline Web services is fully instrumented: each invoked ARM handler either starts or stops an ARM transaction.

The tests for the prototype focused on correct message handling and measurements rather than on processing complex algorithms. When using passive ARM handlers, the processing time for one booking increased by 3.3% in comparison to a run without ARM handlers. Full instrumentation (start or stop of an ARM transaction in every handler, which results in eight measurements per customer request) introduced an overhead of 48.7%; using dummy ARM data in the message's SOAP header in combination with passive ARM handlers increased the

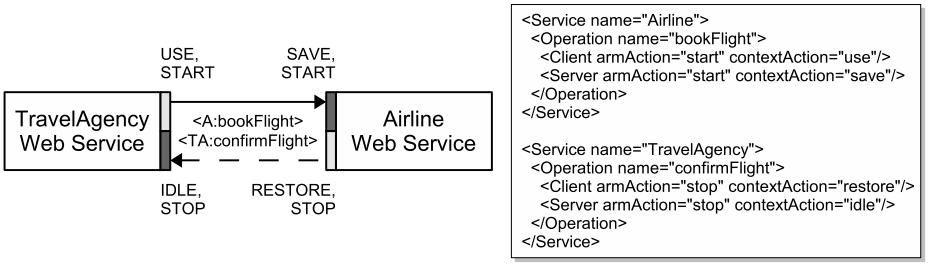

Fig. 6. Configuration Example

response time by 24.7% already. These results show that half of the overhead was caused by message meta-data. For otherwise short SOAP messages, this information increased the message size in a way that response times increased remarkably. However, idle ARM message handlers did not add remarkable overhead to the response time. Of course, the gathered measurement data shows correct dependencies between the executed transactions.

The performance penalty measured in this travel agency scenario is so grave, because the services themselves do not contain complex business logic (which would require more processing time). In addition, the messages exchanged between the Web service partners are rather short. For the presented approach, these circumstances represent the worst-case scenario. In a real-life application with more complex applications and messages, the percentage of the overhead would be lower.

6 Conclusion and Future Work

This paper presented a generic approach for performance instrumentation of synchronous, asynchronous and one-way Web services for end-to-end performance measurements. The approach allows a very fine-grained view upon deployed Web services and allows a user to configure the scope of instrumentation. The runtime information gathered shows the dependencies between Web service invocations and lists the durations of service invocations and instrumentation processing results. The instrumentation approach renders manual source code instrumentation and adaption to specific services unnecessary; the only remaining task is to define (configure) the required measurement points.

The approach relies on standardized JAX-RPC message handlers and message contexts for modifying SOAP messages and adding instrumentation information. It is based on ARM, an instrumentation approach broadly accepted by the industry and supported by large management platforms.

The prototype proves the usability of the approach, although the implementation needs to be optimized. Further steps will be optimization of the prototypical implementation, adaption to additional Web service platforms and instrumentation of a real world Web services-based application environment. In the future,

this approach may provide an easy means for integrating Web services-based applications with management platforms.

Acknowledgements

The author would like to thank Prof. Dr. Reinhold Kroeger from Wiesbaden University of Applied Sciences and Damian McGrath, M.Sc. from IONA Technologies for supervising the work on his diploma thesis, on which this paper is based. He would also like to thank the people at IONA Technologies for the great support during his internship, in which he wrote this thesis.

References

1. The Apache Software Foundation: Apache Axis (Java). (2005) http://ws.apache.org/axis/.
2. IONA Technologies: IONA Artix. (2005) http://www.iona.com/products/artix/.
3. Microsoft Corporation: .NET Framework Developer Center. (2005) http://www.msdn.microsoft.com/netframework/.
4. IBM Tivoli: IBM Tivoli Composite Application Manager for SOA. (2005) http://www.ibm.com/software/tivoli/products/composite-application-mgr-soa/.
5. Computer Associates: Unicenter Web Services Distributed Management. (2005) http://www3.ca.com/solutions/Product.aspx?ID=4714.
6. AMD, Dell, Intel, Microsoft, Sun: Web Services for Management (WS-Management). (2004) http://msdn.microsoft.com/ws/2004/10/ws-management/.
7. Organization for the Advancement of Structured Information Standards (OASIS): OASIS Web Services Distributed Management (WSDM) TC. (2005) http://www.oasis-open.org/committees/wsdm.
8. Schaefer, J.: Methods and Tools for ARM-based Performance Instrumentation of Web Services. Diploma Thesis, Wiesbaden University of Applied Sciences (2005)
9. The Open Group: Application Response Measurement - ARM. (2005) http://www.opengroup.org/arm/.
10. The Open Group: ARM 4.0 Java Language Binding Technical Standard 4.0. (2003) http://www.opengroup.org/arm/uploads/40/3945/C037.pdf.
11. World Wide Web Consortium: Web Service Definition Language (WSDL) 1.1. (2001) http://www.w3.org/TR/2001/NOTE-wsdl-20010315.
12. World Wide Web Consortium: Simple Object Access Protocol (SOAP) 1.1. (2000) http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.
13. Sun Microsystems: Java API for XML-Based RPC Specification 1.1. (2003) http://java.sun.com/xml/downloads/jaxrpc.html.
14. ObjectWeb: Celtix. (2006) http://celtix.objectweb.org/.