TU WIEN
TECHNISCHE UNIVERSITÄT WIEN
VIENNA UNIVERSITY OF TECHNOLOGY

Master's Thesis

The Daios Framework - Dynamic, Asynchronous and Message-oriented Invocation of Web Services

carried out at the

Information Systems Institute
Distributed Systems Group
Technical University of Vienna

under the guidance of
Univ.Prof. Dr. Schahram Dustdar
and
Univ.Ass. Dipl.-Ing.(FH) Florian Rosenberg
as the contributing advisor responsible

by

Philipp Leitner, Bakk.rer.soc.oec.
Sperrgasse 14/15
1150 Wien
Matr.Nr. 0225511

Vienna, 18th September 2007 ____________________________

Abstract

The principle of “publish-find-bind” is one of the cornerstones of Service-Oriented Architectures: service providers publish their services in service registries, service consumers use these registries to find services that they can use, and once a consumer has found an adequate service he can bind and use it. In order to implement this principle Web service clients obviously need to be able to bind to arbitrary services at run-time. Using current client-side service frameworks such as Apache Axis 2, Apache WSIF, Codehaus XFire or Apache CXF it is hard or even impossible to do this - if a dynamic invocation interface exists at all it is often awkward to use and limited in power. Additional problems are introduced by the nature of WSDL: given that WSDL is very much focused on the notion of “operations” it is not surprising that most of the currently deployed Web services also follow a strict “RPC-like” instead of a document-based approach, ultimately leading to tightly coupled architectures instead of SOAs.

This master’s thesis introduces the Daios framework, a client-side Web service framework that overcomes these limitations by (1) providing a dynamic invocation interface that does not rely on precompiled stubs, (2) abstracting from the unseemlinesses of WSDL and exposing a simple and asynchronous messaging interface instead, and (3) supporting the invocation of SOAP/WSDL-based as well as RESTful services through a transparent interface. The thesis details the state of the art in the area of SOA, Web services and REST, gives an overview over relevant related work in the field, explains the design of the framework prototype implemented as part of the practical thesis work, depicts Daios’ usage by means of real-world services, and finally compares the performance of the framework to Axis 2, WSIF, XFire and CXF in terms of supported functionality, runtime performance and memory usage. The evaluation concludes with the result that Daios is on one level with current state of the art Web service frameworks regarding runtime performance, and that the framework is a sound choice for developers facing the dynamic service invocation problem.

Kurzfassung

Das Prinzip “Publish-Find-Bind” ist einer der wichtigsten Eckpunkte von service-orientierten Architekturen: Service Provider veröffentlichen ihre Services in einer Registry, wo diese dann von interessierten Service Consumern gesucht und gefunden werden können. Damit diese auch aufgerufen werden können müssen sie allerdings gebunden werden. Es ist daher unablässig, dass Consumer in der Lage sind, beliebige Services zur Laufzeit zu binden. Mit gegenwärtigen Web Service Frameworks (wie z.B. Apache Axis 2, Apache WSIF, Codehaus XFire oder Apache CXF) ist dies oft nicht möglich oder zumindest nicht einfach - die Interfaces zum Binden zur Laufzeit sind oft umständlich zu benutzen und nicht für alle denkbaren Services anwendbar. Auch die Struktur von WSDL verursacht Probleme: WSDL ist sehr stark auf das Konzept von aufrufbaren “Operationen” zentriert - es ist daher nicht weiter verwunderlich, dass die meisten gegenwärtigen Web Services auch sehr operations- und RPC-zentriert arbeiten. Das Result sind meist sehr stark gekoppelte Anwendungen, die ganz im Gegensatz zu den angestrebten service-orientierten Architekturen stehen.

Die vorliegende Master-Arbeit beschreibt das Daios Framework. Daios ist eine clientseitige Lösung für die oben angeführten Probleme, die (1) ein Interface zum Binden beliebiger Services zur Laufzeit bereitstellt, die (2) die RPC-zentrischen Details von WSDL abstrahiert und statt dessen über ein simples Messaging-Interface (das auch asynchron benutzt werden kann) verwendet wird, und die (3) die Benutzung von SOAP/WSDL-basierten wie auch von sogenannten RESTful Services über ein einheitliches Interface erlaubt. Die Arbeit beschreibt den State of the Art in den Bereichen SOAs, Web Services und REST, gibt einen Überblick über wichtige verwandte Forschung, skizziert das Design des Daios-Prototypen, der als praktischer Teil im Rahmen dieser Master-Arbeit entstanden ist, und geht auch kurz auf dessen Verwendung ein. Schließlich wird der Prototyp mit existierenden Projekten (Axis 2, WSIF, XFire und CXF) in Bezug auf Funktionalität, Laufzeit und Memory-Verbrauch verglichen. Diese Evaluierung zeigt, dass das Daios-Framework in Bezug auf Performance auf ähnlichem Niveau ist wie die besten etablierten Service Frameworks, und daher für Entwickler, die vor den o.g. Problemen stehen, eine gute Wahl darstellt.

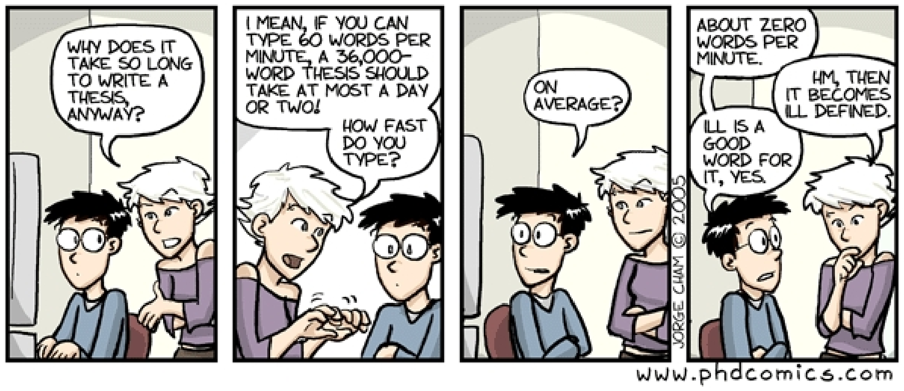
Danksagung

Die vorliegende Magisterarbeit stellt das (glückliche) Ende fünf meistens arbeitsreicher, immer jedoch schöner Studienjahre der Fächer Wirtschaftsinformatik und Informatik in Wien dar. Viele wichtige Menschen haben mich durch diese Zeit oder einen Teil davon begleitet.

Die Wichtigsten waren mit Bestimmtheit meine Eltern, Monika und Wolfgang Leitner. Euch möchte ich hiermit meinen aufrichtigen Dank aussprechen. Ohne eure Bestärkung, Bestätigung, finanzielle und auch anderweitige Unterstützung wäre mein Studium in dieser Form nicht denkbar gewesen. Ein ebenso großer Dank gebührt auch den Betreuern dieser Arbeit, Florian Rosenberg und Schahram Dustdar. Bessere Betreung als ich sie vorgefunden habe kann sich ein Diplomand nicht wünschen.

Einen besonderen Dank möchte ich an dieser Stelle auch an Martin Zach und das Team der Siemens PSE SMC richten, da mir Siemens die einmalige Gelegenheit bot, bereits während meines Magisterstudiums in die Welt der Forschung einzutauchen und "echte" wissenschaftliche Arbeit im Rahmen eines internationalen Projekts zu verrichten. Zum Erfolg dieser Arbeit trugen aber auch noch andere bei: besonders möchte ich mich noch bei Philipp Glatz und Martin Treiber für ihren Input zu verschiedenen Versionen dieser Magisterarbeit bedanken. Martin war es auch, der mich überhaupt erst zum Verfassen meiner Abschlussarbeit am Institut für Informationssysteme motiviert hat.

Und schließlich gilt es noch ein ganz dickes "Dankeschön" an meine *wichtigste* Reviewerin auszurichten: vielen, vielen Dank an Julia Karrer für alle Unterstützung beim Verfassen dieser Arbeit, wie auch in allen anderen Dingen. Je t'embrasse $\ddot\smile$


WHY DOES IT TAKE SO LONG TO WRITE A THESIS, ANYWAY?
I MEAN, IF YOU CAN TYPE 60 WORDS PER MINUTE, A 36,000-WORD THESIS SHOULD TAKE AT MOST A DAY OR TWO!
HOW FAST DO YOU TYPE?
ON AVERAGE?
JORGE CHAM © 2005
ABOUT ZERO WORDS PER MINUTE.
HM, THEN IT BECOMES ILL DEFINED.
ILL IS A GOOD WORD FOR IT, YES.
www.phdcomics.com

Contents

1 Introduction 10

1.1 Motivation 11

1.2 The Daios Solution 13

1.2.1 Requirements 13

1.2.2 Daios 14

1.3 Organization of this Thesis 15

2 State of the Art Review 17

2.1 Distributed Object Middleware 17

2.1.1 General Concepts 18

2.1.2 Dynamic Invocation Interfaces 18

2.2 Service-Oriented Architectures 19

2.2.1 Definition and Concepts 19

2.2.2 Advantages 22

2.3 Web Services 22

2.3.1 SOAP 24

2.3.2 WSDL 26

2.4 REST 28

2.4.1 Interface Description in REST 30

3 Related Work 32

3.1 Dynamic Service Invocation 32

3.1.1 WSIF 32

3.1.2 Apache Axis 2 34

3.1.3 JAX-WS 35

3.1.4 Other Java Frameworks 36

3.1.5 SOAP::Lite 36

3.1.6 Dynamic Invocation using XML Type Subsumption 37

3.1.7 Design Patterns for stubless Service Invocation 38

3.1.8 Other Approaches 39

3.2 Message-based and Asynchronous Service Invocation 40

3.2.1 Client-side Asynchrony Patterns 41

3.2.2 SSDL 42

3.2.3 WSMQ 43

4 Design and Implementation 46

4.1 Architecture 47

4.1.1 Frontend 48

4.1.2 Daios Messages 50

4.1.3 Interface Processing 52

4.1.4 Service Invoker 52

4.2 Implementation . . . 56
4.2.1 Implementation Issues . . . 57
4.3 Client-Side Interface . . . 60
4.4 Advanced Features . . . 62
4.4.1 Intercepting the Framework . . . 62
4.4.2 Fixing Daios Behavior in non-standard Situations . . . 63

5 Evaluation 66
5.1 Evaluation Scenario . . . 67
5.2 Detailed Analysis . . . 68
5.2.1 Internal Processing . . . 68
5.2.2 Memory Consumption . . . 70
5.3 Comparison to other Frameworks . . . 71
5.3.1 Functional Analysis . . . 71
5.3.2 Comparison of Runtime Performance . . . 74
5.3.3 Memory Consumption . . . 77
5.4 Evaluation Summary . . . 78
5.4.1 Limitations . . . 79

6 Conclusion and Future Work 80
6.1 Future Work . . . 81

A List of Abbreviations 84

B SOAP RPC Example 86

C Complete WSDL Example 87

D Complete SSDL Example 89

E Structural Distance Calculation 90

F REST by Example Procedure 91

G WSDL Description to Listing 11 92

H Input Stream Data Source used in Daios 93

I Complete Daios SOAP Example 94

J Complete Daios REST Example 95

K A Logging Interceptor for Daios 96

L Performance Comparison Results 97

List of Figures

1 A timeline of distributed computing . . . 10
2 The triangle of publish-find-bind . . . 21
3 The Web service standards stack . . . 24
4 Structure of a SOAP message . . . 25
5 Multihop transmission of a SOAP message . . . 25
6 CPWSI in UML notion . . . 39
7 Service invocation using reflection . . . 40
8 WSMQ overview . . . 44
9 General dynamic invocation procedure . . . 47
10 Daios overall architecture . . . 48
11 Daios Frontend Structure in UML syntax . . . 49
12 Dynamic SOAP invocation . . . 53
13 Structural distance example . . . 55
14 Dynamic REST invocation . . . 55
15 REST by example . . . 56
16 AXIOM XML model . . . 58
17 Daios runtime performance - SOAP invocation . . . 68
18 Daios runtime performance - REST invocation . . . 69
19 Daios memory consumption - SOAP invocation . . . 70
20 Daios memory consumption - REST invocation . . . 71
21 Comparison of preprocessing costs . . . 75
22 Comparison of simple invocations - linear regression . . . 76
23 Comparison of array invocations - linear regression . . . 76
24 Comparison of binary invocations - linear regression . . . 77
25 Comparison of maximum memory consumption . . . 78
26 Comparison of simple invocations . . . 97
27 Comparison of array invocations . . . 97
28 Comparison of binary invocations . . . 97

List of Tables

1 Advantages of SOA 22
2 Client-side asynchrony patterns 42
3 Factory class names of Daios backends 49
4 Utility components in the Daios framework 50
5 Atomic type mapping in Daios 51
6 Third-party software used in Daios 57
7 Interceptable events in Daios 63
8 Web service standards support 72
9 Dynamic invocation features 73
10 List of Abbreviations 85

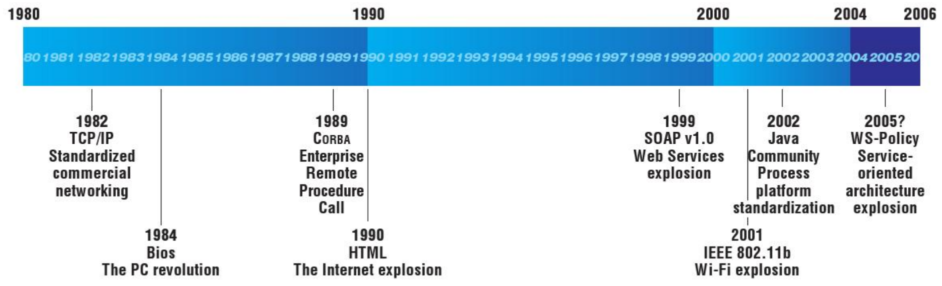
List of Listings

1 Integer stack implemented in CORBA . . . 18
2 WSDL extensibility . . . 27
3 WADL example . . . 30
4 Dynamic invocation in WSIF . . . 33
5 Type mapping in WSIF . . . 33
6 Dynamic invocation in Axis 2 . . . 34
7 JAX-WS Proxy example . . . 36
8 SOAP invocation in Perl using SOAP::Lite . . . 37
9 SSDL protocol definition . . . 43
10 Frontend creation in Daios . . . 50
11 Constructing Daios messages . . . 51
12 Parsing types in Daios . . . 59
13 Using XMLBeans to parse WSDL . . . 60
14 Blocking invocation in Daios . . . 60
15 Hard-coding WSDL parameters in Daios . . . 63
16 Accessing the invocation backend in Daios . . . 64
17 SOAP and HTTP headers in Daios . . . 65
18 SOAP-encoded RPC call . . . 86
19 Complete WSDL Example . . . 87
20 Complete SSDL example . . . 89
21 Structural distance calculation in pseudo-code . . . 90
22 REST by example algorithm . . . 91
23 WSDL description corresponding to Listing 11 . . . 92
24 Input Stream Data Source for AXIOM . . . 93
25 Complete example of Daios SOAP invocation . . . 94
26 Complete example of Daios REST invocation . . . 95
27 Logging SOAP payload using a Daios interceptor . . . 96

1 Introduction

One Ring to rule them all, One Ring to find them,
One Ring to bring them all and in the darkness bind them.
In the Land of Mordor where the Shadows lie.
– J.R.R. Tolkien, "The Lord of the Rings" [79]

Ever since computers changed from being huge number crunching engines to smaller "personal" computers in the mid-eighties users had the wish to be able to connect several of these PCs to form a single, more capable processor. This need was fulfilled with the dawn of high-speed LAN and WAN (local and wide area networks): computers were now able to communicate and cooperate to execute tasks that none of them could handle on its own. Computers were finally freed from their isolation and were grouped into computer networks [77]. In the wake of this revolution *distributed systems* emerged as a new way of building complex software. Distributed systems are usually defined as software systems that consist of n (with $n > 1$) physically independent computers, but look like one single coherent system to the user [77].



1980
1990
2000
2004
2006
1982 TCP/IP Standardized commercial networking
1984 Bios The PC revolution
1989 CORBA Enterprise Remote Procedure Call
1990 HTML The Internet explosion
1999 SOAP v1.0 Web Services explosion
2001 IEEE 802.11b Wi-Fi explosion
2002 Java Community Process platform standardization
2005? WS-Policy Service-oriented architecture explosion

Figure 1: A timeline of distributed computing, taken from [49]

Distributed systems in varying shapes have had a huge impact on enterprise information technology from then on: newly implemented systems have almost always been built with distribution in mind, and existing ("legacy") applications were integrated into company IT networks. The idea of *enterprise application integration* (EAI) [54] was born. Unfortunately EAI proved to be much harder than people had expected: legacy systems were built on a huge number of different hardware and software platforms, they were using various proprietary protocols and the number of applications that had to be integrated was growing almost on a daily basis. These difficulties gave rise to a number of distribution and integration technologies: *message-oriented middleware* [8, 47] (MOM) provided flexible integration patterns such as Publish/Subscribe [30], and middleware systems for *remote procedure calls* [10] (RPC) massively simplified the task of writing distributed software systems. With the wide adoption of the *object-oriented programming paradigm* during the nineties RPC middleware was quickly expanded to *distributed object middleware* [77], providing the ability to (apparently) call objects on remote machines as if they were standard in-memory objects.

Despite all those achievements a few problems remained: the components that formed distributed systems were indeed physically distributed, but logically still quite tightly coupled. Replacing one of these components was about as hard as replacing a component in a standalone application. Additionally few of these middleware systems were really platform-independent, introducing new problems as soon as computers operating on different technological platforms (such as processors, operation systems, programming languages, ...) had to be integrated. *Service Oriented Computing* [62] (SoC) aims at eliminating these issues: components in SoC (prevalently referred to as *services*) are dynamically mashed up using standardized interfaces and protocols. Distributed systems in SoC are therefore composed of services that may be built on various platforms, and which may be exchanged easily. For all this SoC relies on a special architecture, the *Service-Oriented Architecture* [61] (SOA). Two technologies have been established as the standard way to build SOAs: widely known are *Web services* [43, 80] and the representational state transfer (REST) [33, 65] model. SOA, Web services and REST will be explained in more detail in Chapter 2 of this thesis.

1.1 Motivation

As Chapter 2 will explain, Service-Oriented Architectures provide loose coupling of service providers and service consumers by utilizing the triangle of the operations "publish", "find" and "bind" [61]. Producers *publish* their services using a standardized interface language by registering it in a *service repository* (or service registry). Consumers can then use this repository to discover (*find*) registered services and *bind* them in order to be able to invoke these services.

The first two parts of this triangle, publish and find, particularly put requirements on the service registry and the interface definition language: in order to be able to publish services an expressive and extensible service definition language has to be available and be supported by the service registry. Find (look-up) demands for a reasonably expressive query language in the registry that allows service consumers to discover services in an automated way. Currently, no *Web service Registry* implementation can fully live up to these demands. UDDI [81] does not provide a sufficient query interface, while the ebXML registry [38] seems powerful but also overly complex [26, 45].

The third operation, bind, is independent from the service registry: binding has to be handled solely by the service consumer. It is essential for the success of SoC that the consumer can connect to any service that he might discover during the find step without troubles, and that it is possible to change this binding at any time (specifically at run-time of the system). As Chapter 3 will summarize this is not easy with current state of the art Web service client frameworks such as Apache Axis 2 [4] or Apache WSIF [6]. These frameworks heavily rely on *client-side stubs* to invoke services. Stubs are usually autogenerated and make the actual Web service call very easy (almost transparent) for the developer, but they are invariably hardwired to a specific service provider and cannot be changed at run-time. Actually the service provider cannot even be changed at compile-time, since a re-generation of the Web service stubs as well as a redesign of the client application has to take place as soon as the provider changes. This is a severe problem for realizing a SOA: if service providers are hardwired into the service consumer's application code producers and consumers cannot by any means be considered loosely coupled. The usage of client stubs does not follow the idea of SOA, since find as well as bind are in such a situation actually carried out by the developer: he (as opposed to the client application itself) decides which Web service should be used to provide a specific service, and he (again as opposed to the client application) binds the client to this service. An application which relies on client-side stubs cannot realize a SOA as defined in Chapter 2.

What is needed for the realization of "publish-find-bind" using Web services is a framework that allows for "stubless" service invocation, that is invocation of services without depending on any kind of precompiled service access components. Such a *dynamic service invocation* framework could (as the SOA vision as described in Chapter 2 expects) bind to any Web service at run-time, and re-bind at any given time without the need to recompile or redesign the client application. With existing frameworks such a stubless call is usually possible using low-level functions or APIs (what is obvious, since the generated client-side stubs also need some way to execute the actual Web service call), but often this functionality is overly hard to use and restricted in power. WSIF for example provides a *dynamic invoker*, but it is by default not capable of calling arbitrary Web services. Only services that do not rely on *complex types* as message parts are supported. This limitation does not seem acceptable for a real-life application. Apache Axis 2 on the other hand provides a fully expressive dynamic invoker, but offers only very limited support to the client-side application developer when constructing dynamic invocations.

Additionally, existing Web service client frameworks as they are described in Chapter

3 often suffer from a few further misconceptions. They are often built to be as similar as possible to earlier distributed object middleware systems [83], implying a very strong emphasis on RPC-centric and symmetric Web services. The reasons for this are manifold. First of all, developers are very accustomed to RPC-style development, while substantially different communication paradigms (for instance message-driven or space-based computing [42, 53]) were (and still are) having a hard time to catch on. It is therefore only natural that Web services (which offer the possibility to be either used in the document-centric style of SoC or in RPC-style) are much more often used RPC-centric. This is further supported by the structure of WSDL [84], the dominant standard for Web service interface definition. As [63] puts it: "WSDL's focus on an interface abstraction for describing services makes it difficult to change the object-oriented or Remote Procedure Call (RPC) mindset and focus on message-orientation and asynchrony (...)". WSDL is therefore somewhat suboptimal for a SOA which is based on the exchange of business documents.

1.2 The Daios Solution

1.2.1 Requirements

Taking into account the fundamental maladies of currently available Web service client-side solutions one can define the following requirements for a *Web service invocation framework* that truly supports the SOA vision:

- *Stubless service invocation*: Given that generated stubs entail a tight coupling of service provider and service consumer the invocation framework shall not rely on any pre-generated and precompiled components such as client-side stubs. Instead the framework should be able to invoke any Web service through a single interface.
- *Protocol-independent*: Web service standards and protocols have not yet fully settled. There is still ongoing discussion about the advantages of the REST architecture as compared to the more common SOAP approach to Web services. The invocation framework should therefore be able to abstract from the underlying Web service protocol, and support at least SOAP-based and REST-based Web services transparently.
- *Message-driven*: Currently Web services are often seen as a collection of platform-independent remote methods. The framework shall be able to abstract from this RPC style which usually leads to tighter coupling and follow a message-driven approach. The framework shall take an input message from the user, and return an output message. The user of the framework shall not be concerned

with what the interface (the WSDL description) of the actual Web service looks like more than what is semantically necessary. Hence the user shall not need to know about operations, port types, ports, and bindings of the invoked service.

- *Support for asynchronous communication*: In a SOA services might take a long time to process a single request. The currently prevalent request/response style of communication is not suitable for such long-running requests. The framework shall therefore also support asynchronous (non-blocking) communication.
- *Simple API*: Current dynamic invokers are often not intuitive to use. The framework shall utilize the message-driven approach to make the API to the user as simple as possible. The API shall make intensive use of optional parameters. All details of the Web service call shall be definable by the user, but as few as possible shall be mandatory. Therefore the API of the framework shall be powerful enough for the experienced, and simple enough for the casual user.
- *Acceptable runtime behavior*: The framework shall not imply sizable overhead on the Web service invocation. Using the framework shall not be significantly slower than using any of the existing Web service frameworks.

1.2.2 Daios

With the requirements presented in Section 1.2.1 in mind a new client-side Web service framework was designed and implemented. Daios (Dynamic and asynchronous invocations of services) works as a dynamic invoker for any Web service that is accessible either via the SOAP [85] protocol or through a REST-based Web service interface. Daios supports WSDL as interface definition language for SOAP-based services and "REST by example" (see Chapter 2) for REST-based services.

Daios is designed to be used similar to MOMs: the client developer is not supposed to "call operations" of the target service, but to simply pass a message to it. Daios will extract the necessary low-level information (such as operation name, port and port type, binding, . . .) from the interface definition, dynamically construct a call and return the result to the client (again as a message). Additional low-level parameters (for instance SOAP headers) may be set in order to force Daios to exhibit some specific behavior. Note that this approach decouples the Daios-powered client application from the service interface: even if the description of the service changes (but the semantics stay the same) the necessary Daios client code may still remain unchanged.

Message-passing systems are inherently asynchronous. Daios therefore supports (besides the standard synchronous request-response style) three different *invocation asynchrony patterns* [82] for asynchronous communication: “Fire and Forget” is used in cases when no response is expected from the service provider; “Poll Object” and “Callback” both return a result as the service finished. “Poll Object” allows the client to decide on its own when it wants to process the result, while “Callback” immediately interrupts the client to return the result.

Chapter 4 will present the design of the Daios framework in detail, and give code examples demonstrating how Daios can be used in real life. Chapter 5 will give a critical evaluation of the implementation of the framework, compare its performance against existing Web service technologies, and comment on whether the goals that lead the Daios design and implementation have been met.

1.3 Organization of this Thesis

The rest of this thesis is organized as follows:

Chapter 2 will detail the current state of the art in distributed object middleware and Service-Oriented Architectures. It will explain the technological background of Web services and show how they can be utilized to build SOAs. The Chapter will be concluded with a look at the representational state transfer (REST) model which is the basis of the WWW (World Wide Web), and can also be applied to SOA.

Chapter 3 will then continue with a consideration of the relevant related work in the area of Web service invocation. It will show a selection of client-side solutions that are available at the moment, and evaluate how useful they are for dynamic service invocation. Special emphasis will be put on solutions in the Java programming language, but other solutions will also be mentioned.

The design as well as the actual implementation of Daios will be described in Chapter 4. This Chapter will explain the overall architecture of Daios in some detail and will give some code examples of how Daios can be used in practice.

Chapter 5 will provide a critical evaluation of the Daios framework. It will compare Daios to existing solutions in terms of functionality, invocation response times and memory consumption.

Finally Chapter 6 will conclude the thesis with some final remarks and an overview of future work.

2 State of the Art Review

"What does your architecture look like?" - "My architect thinks it's service-oriented, my developers insist it's object-oriented, and my analysts wish it would be more business-oriented. All I can tell is that it is not what it was before we started using Web services."
– Discussion at a "SOA-meets-business" conference [29]

The following Chapter will detail the current state of the art in the area of distributed computing with emphasis on distributed object middleware, Service-Oriented Architectures and Web service technologies. This Chapter is intended to explain the technologies, concepts and notions that will be used through the rest of this thesis.

2.1 Distributed Object Middleware

In Chapter 1 the idea of operations that can be invoked remotely over a network across the boundaries of memory spaces and physical computers (RPC invocations) has been introduced . Distributed object middleware systems are a direct successor of the early RPC-based systems: they combine the idea of remote invocation with the notions of object-oriented programming (OOP) [92]. Early research systems such as Arjuna [64] and Emerald [11] for the C++ language were not widely adopted by practitioners for various reasons and are not in use anymore. The first industry-strength (and still most important) distributed object systems were OMGs CORBA (Common Object Request Broker Architecture), Microsoft's DCOM (Distributed Component Object Model) and the RMI (Remote Method Invocation) functionality implemented in Java. The development of these systems has been driven by the wish to bring the strengths of OOP such as object references, inheritance, exception handling and polymorphism to distributed systems [28]. The general idea of all these systems is to provide a unified view on local and remote objects [90]. This intend has not been fully accomplished: today there are still fundamental differences between local and remote invocations regarding latency, concurrency, error handling and memory access [90]. Schmidt et al. have therefore dedicated the second as well as the fourth book in the famous POSA (Pattern-oriented Software Architecture) series to patterns for distributed computing [13, 69], describing best practices to minimize the evident challenges of distibuted computing.

2.1.1 General Concepts

One important concept of distributed object middleware is the notion of *interface definition* through an IDL (Interface Definition Language). The IDL describes the public interface of a remote object in the same way a class definition describes the interface of a local Java object. Listing 1 depicts an example of an IDL definition of a integer stack implemented in CORBA. The similarities to interface definitions in object-oriented programming languages are rather obvious.

```
interface CORBAIntStack : Object {
  exception Overflow{};
  exception Underflow{};
  void push (in int newVal) raises (Overflow);
  int pull () raises (Underflow);
};
```

Listing 1: Integer stack implemented in CORBA

Other techniques used to reduce the complexity of distributed object middleware systems include the use of *client* and *server stubs*. Stubs are "generated by remote procedure call systems in order to implement type-specific concerns, such as marshalling and operation dispatch. The input to that generation process is an interface definition that defines the formal parameters (. . .)" [28]. Usually, middleware systems include a tool that compiles stubs from the according IDL. Stubs provide *static type safety* to distributed programming: invocations using the client stubs are type safe as long as the server interface (the IDL interface of the server) does not change.

2.1.2 Dynamic Invocation Interfaces

Although statically generated stubs exhibit distinct advantages they also impose some restrictions on the applications developed: most importantly these stubs imply a tight (design-time) coupling between client and server objects. Since distributed object middleware treats local and remote objects identically this coupling is as tight as between local objects; an interface change in one of the objects will most likely cause all implementations to change that depend on that object. For most application domains this restriction is not too bad, but there are problem areas where a looser coupling is desireable or necessary.

Distributed object middleware systems therefore often specify a *dynamic invocation interface* (DII). The DII allows for "on-the-fly" requests to objects not yet known during design-time of the application. Such a dynamic request has to define at least the following items [28]:

- The server object that should be invoked.
- The name of the operation that should be invoked.
- The parameters of the invocation.
- A data structure that is made available for the result.

Dynamic invocation is never type safe. It can neither be ensured that the requested server object actually exists, nor that it exports the requested operation, nor that the parameters given fit the operation.

Dynamic Invocation Interface is also the name of a pattern in the fourth POSA book [13], which describes (in an abstract form) dynamic interfaces to various types of components. DIIs of distributed object middleware systems are concrete implementations of this pattern.

2.2 Service-Oriented Architectures

Service-Oriented Architectures (SOAs) are one of the latest “buzzes” in the fast moving computer science industry, getting a lot of attention from researchers as well as from practitioners. SOAs are considered as “the next major step in distributed computing” [61] by a big part of the research community today.

2.2.1 Definition and Concepts

Despite this tremendous advertence in the field an universally accepted definition of what actually makes a SOA has not emerged so far. The Organization for the Advancement of Structured Information Standards (OASIS) has recently published a reference model for Service-Oriented Architectures [39] which defines a SOA as follows:

> Service-Oriented Architecture (SOA) is a paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. [39]

Unfortunately this definition is per se not very satisfying: it is so general that it can be applied to almost any architecture for heterogeneous systems. Another definition was stated by Thomas Erl in [29]:

> SOA is a form of technology architecture that adheres to the principles of service-orientation. When realized through the Web service technology platform, SOA establishes the potential to support and promote these principals throughout the business process and automation domains of an enterprise. [29]

This definition is (very much like the OASIS definition) not very rich in content: SOA is more than just an architecture built on top of services. The following Section will try to highlight the defining features of any SOA in a more explicit way.

First of all, Service-Oriented Architectures are centered around *services.* According to [29] services are:

- *loosely coupled* - services are self-contained and self-managing. The number of necessary connections to systems "outside" of the service are minimal. Services have low *representational*, *identity* and *communication protocol coupling* [59].
- *defined by a service contract* - services adhere to a communications and interface definition or to a service description,
- *autonomous* - services have the absolute control over the function that they realize,
- *abstract* - services hide all implementation details from the rest of the world, revealing only the service contract,
- *reusable* - services are intended for and promote reuse,
- *composable* - in order to promote reuse services are easily composable, i.e. simple services can be assembled and coordinated to build composite services (service composition) [24, 48],
- *stateless* - services do not have a state, and
- *discoverable* - services can be found and evaluated via external discovery or registry mechanisms.

As already pointed out, services are an important, yet not the only feature that defines a SOA. One other concept is from eminent importance in that context: Figure 2 shows what will be referred to as the "triangle of *publish-find-bind*" throughout the remainder of this thesis.

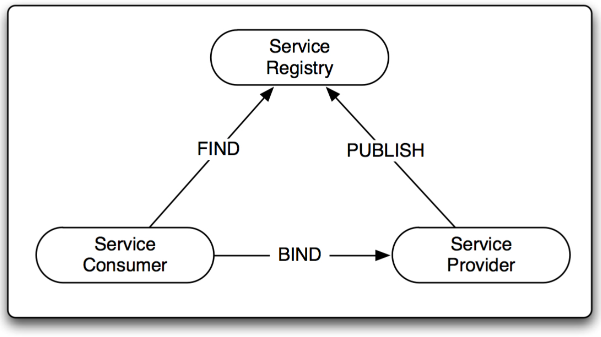


Figure 2: The triangle of publish-find-bind

Figure 2 shows that in any SOA three distinct roles have to be present:

- The *service consumer* (or service requester, service client) is interested in a certain capability. He needs a service that provides that capability.
- The *service provider* (or service implementer) is able to provide a certain capability. He needs a client that he can serve (and, usually, charge).
- The *service registry* (or service broker, service discovery agency) is the broker that brings the former together. The service registry knows which provider has which capabilities, and can be queried by the client.

A typical SOA will often consist of a huge number of service requester and service providers, while there will usually be only one service registry (although the registry may be replicated or distributed for reasons of safety and performance).

Requester, providers and registry interact via three different operations: first of all, providers *publish* their services with the service registry, so that requesters can *find* these services. Once a requester has found a service that exposes a capability that he needs he can *bind* to this service. Binding is independent from the registry - as opposed to a *broker* in the POSA pattern [14] the service registry in a SOA is not responsible for mediating the actual service invocation. Note that the registry has a distinct duty in this scenario: it decouples service consumer and service provider. Without the service registry consumer and provider have to know each other right from the start, and can therefore not be considered decoupled *in space* after the categorization in [30].

Another important property of SOAs is the adherence to *open standards*. “Standards are the only way”, as [49] puts it. In a SOA all operations (publish-find-bind) as well as all used basic technologies (registry, service definition, ...) are defined in formats which are freely available to anyone interested.

2.2.2 Advantages

Property	**Advantage**
Based on independent services	Complex systems can be composed of atomic services through service composition
Publish-Find-Bind	Service Registries decouple consumers and providers; consumers can rebind to different providers at run-time; “late configuration” is possible [48]
Based on open standards	Open standards provide interoperability; no “lock in” on vendors or platforms
Coarse granularity	Planned on a high level of abstraction [48]; implementation details not visible outside of the services

Table 1: Advantages of SOA

Table 1 summarizes the properties of a Service-Oriented Architecture, and describes what the advantages of this approach are. Simplifying it can be said that SOAs realize loose coupling of service consumer and service provider by utilizing the triangle of Publish-Find-Bind, and are interoperable by nature through relying on open standards for all communication. Services can be reused easily through service composition. Such an “ideal” SOA will be referred to as the *SOA vision* in this thesis. The term Service-oriented Computing (SoC) [48, 61, 62] has been established for building a distributed system on top of such a Service-Oriented Architecture.

2.3 Web Services

One possibility to realize a Service-Oriented Architecture as defined in Section 2.2, and the one that gets by far the most attention from the scientific community, are *Web services* [23, 43, 80]. A Web service is “a self-describing, self-contained, modular application accessible over the Web. It exposes an XML interface, it is registered and can be located through a Web service registry" [80]. Web services are all about standards: they use SOAP [85] as communication protocol and WSDL, the Web Service Definition Language [84], as interface definition language. SOAP as well as WSDL are independent from programming languages or hardware platforms, and can be processed by any system that is able to process XML, the eXtensible Markup Language.

These attributes align Web services as a robust technology to realize the SOA vision. Revisiting the defining features of SOA services from Section 2.2 one can see how they map to Web service features:

- *loosely coupled*, *autonomous*, *abstract* and *stateless* - Web services are by definition self-contained and modular, and are therefore suitable for building loosely coupled and autonomous services.
- *defined by a service contract* - Web services have interfaces defined in WSDL.
- *composable* and *reusable* - Web services can be composed and choreographed using standardized technologies (for instance WS-BPEL [40]).
- *discoverable* - Web services can (at least in theory, see Chapter 1) be discovered through Web service registries (e.g., UDDI [81] or the ebXML registry [38]).

This strong match between SOA and Web service features has often led people to the misconception that SOA and Web services are more or less interchangeable terms [29, 83], so that every application that is implemented using Web services is also magically service-oriented, or that SOAs can only be built using Web service technology. Nothing could be farer away from the truth: many of today's Web service applications are as strongly coupled as any CORBA or EJB application, and many people in the REST community argue that REST is a much better way of implementing SOAs than SOAP-based Web services. It is an important message to keep in mind that it is not sufficient to just use Web services in order to build a SOA, but to use them in concordance with the SOA paradigm.

Web services are built on top of open standards: the enabling technologies for Web services are XML as data encoding language, and XML Schema for the type system. HTTP is often used as transport protocol, but other bindings (e.g., raw TCP, SMTP, JMS, . . .) are also possible. Web services are often stated to utilize a set of three *core standards* (SOAP, WSDL and UDDI) [23, 80] and optionally an almost unlimited number of extensions (usually referred to as WS-* standards). Figure 3 shows the

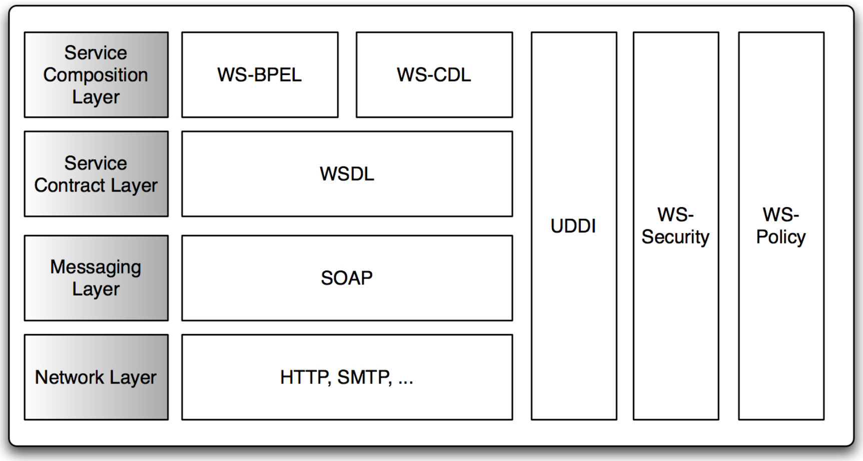


Figure 3: The Web service standards stack

Web service standards stack [24, 43]: on *network level* a transfer protocol (which is independent from Web service technology) is necessary, a usual pick is HTTP but others are possible; on top of this transfer protocol a messaging protocol (SOAP) is defined; above the messaging layer the service definition layer is located, which is implemented in WSDL; at the topmost logical level reside the Web service composition standards, WS-BPEL and WS-CDL [87]. Besides this layered structure a few "vertical silos" are necessary: standards as UDDI, WS-Security and WS-Policy [67] span many or all levels of the Web service stack. For reasons of brevity the WS-* standards are not further covered in this thesis, but there are a lot of good books (for instance [29, 67]) available which dedicate a lot of space to WS-*.

2.3.1 SOAP

SOAP is the communication protocol utilized by Web services. It is a dialect of the XML data encoding standard, and is therefore naturally platform-independent, reasonably human-readable and machine-processable. SOAP was initially created by Microsoft, and further advanced in a joint effort of Microsoft, IBM, Lotus, Developmentor and UserLand [23]. Now it is a recommendation of the World Wide Web Consortium (W3C). Originally SOAP was an acronym for "Simple Object Access Protocol", but with version 1.2 of the SOAP specification [85] it is considered a standalone term [29]. The obvious reason is that the protocol is no longer used to access "remote objects", which made the name rather misleading.

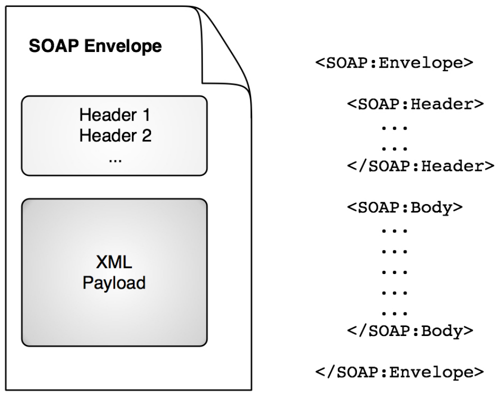


Figure 4: Structure of a SOAP message

Structure. SOAP messages have a simple structure (Figure 4): they consist of an *envelope*, an optional message *header* and a mandatory message *body*. The header contains information that is important for transmitting, relaying and processing of the message, while the body contains the actual XML payload data. SOAP is defined in a very extensible way. Additional headers can be introduced to support new functionality (such as security, transactions, ...). Usually header fields which are unknown to the processor of a message are simply ignored, but the `mustUnderstand` attribute can be used to define headers that have to be interpreted by the processor.

Although the idea of SOAP (as of version 1.2) is to transport a payload *message* it was also a distinct design goal to make the encoding of RPC calls as straightforward as possible [85]. Appendix B shows an example of a message that wraps an RPC call to the operation `getAddress`, with five arguments (`in0` to `in5`). This message contains an empty message header.

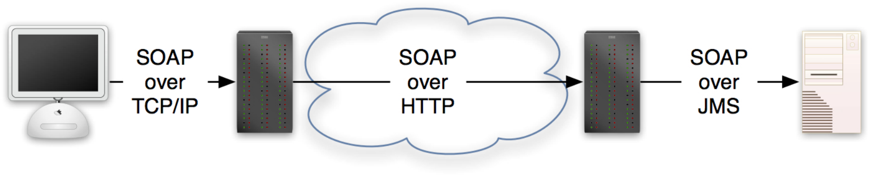


Figure 5: Multihop transmission of a SOAP message

Processing Model. SOAP assumes that it is not always (and with any transport protocol binding) possible to presume a direct point-to-point connection between sender and receiver of a message. Therefore a *processing model* is defined, which can be utilized to transfer messages over multiple hops. The processing model distinguishes three different *roles*. In any communication there is exactly one *initial sender* which creates the message, any number of *intermediaries* which simply forward the message and exactly one *ultimate receiver*, which finally consumes the message. Transmission over multiple hops does not need to use the same transport protocol for every hop. Figure 5 shows an example transmission over three hops using three different transport protocol bindings.

2.3.2 WSDL

Another important W3C recommendation is the Web Service Definition Language (WSDL) [84, 89]. WSDL is used to define the *abstract* and *concrete* interface of Web services. Like SOAP it is an XML grammar and therefore inherently platform-independent. It specifies all important service information, such as what the Web service does, where it is located and how it can be invoked [80]. Very important in WSDL is the separation of abstract definitions from the concrete network deployment and binding details. In its function as interface description language WSDL has some obvious similarities to the IDL used in distributed object middleware (as described in Section 2.1). As Chapter 1 explains, this similarity gives rise to some problems and misconceptions, which are examined within the scope of this thesis. WSDL currently comes in two slightly different flavors: version 1.1 [84] is not recent anymore, but it is broadly accepted in the community and supported by almost all tools on the market. The recent WSDL specification is version 2.0 [89], which brings some enhancements and simplifications, but which is not yet widely supported. If not stated otherwise the information below applies to the 1.1 version of WSDL, since this version is by now much more relevant in the industry.

Structure. The WSDL 1.1 standard defines six major elements [84] that make up any WSDL description:

- *types* provide data type definitions using XML Schema,
- *messages* represent abstract notifications that the service accepts or sends,
- *portTypes* are abstract operations,
- *bindings* bind portTypes to a concrete protocol and data format specification,
- *ports* can be considered as the "endpoints" of bindings, i.e. the address of a certain binding, and

- *services* are sets of ports, i.e. they group a number of related ports.

WSDL 2 renames portTypes to "interfaces", and removes the concept of messages entirely. Messages in WSDL 2 are specified directly in XML Schema [89]. A full example of a WSDL 1.1 definitions file which uses all of the above major elements is provided as appendix C.

One advantage of WSDL is the easy extensibility of the language. The WSDL standard itself [84, 89] uses extensibility to provide language bindings, for instance for SOAP and HTTP. Listing 2 shows how extensibility (the element `wsdlsoap:address`) is used to define a SOAP endpoint.

```
<wsdl:service name="MessageBasedOrderService">
  <wsdl:port binding="impl:OrderServiceSoapBinding"
             name="OrderService">
    <wsdlsoap:address location="http://localhost/OrderService"/>
  </wsdl:port>
</wsdl:service>
```

Listing 2: WSDL extensibility

MEPs. As a legacy of distributed object middleware WSDL endpoints are often invoked in a request/response manner. However, this is not the only *transmission primitive* (Message Exchange Pattern, MEP) supported [84]:

- *One-way* - the endpoint receives a message, and does not reply.
- *Request-response* - the endpoint receives a message, and replies with a message.
- *Solicit-response* - the endpoint sends a message, and receives a response.
- *Notification* - the endpoint sends a message, and does not expect a reply.

WSDL 2 further extends the support for MEPs: in WSDL 2 eight MEPs are predefined, and further MEPs can be introduced easily.

WSDL Binding Styles. WSDL is so frequently used in conjunction with SOAP that the WSDL recommendation predefines two different styles of WSDL-to-SOAP binding. The two possibilities are *RPC style* or *document style* [15]. Both of these encoding styles can have a so-called use, either *encoded use* or *literal use.* This sums up to four different combinations: `RPC/encoded`, `RPC/literal`, `document/encoded` and `document/literal`, all of them mutually incompatible. To clean up this mess an industry organization (the "Web services interoperability organization", WS-I) was formed. WS-I has released a *basic profile* that defines interoperable Web services [91]. The WS-I basic profile basically bans the encoded use for its interoperability issues, and promotes `document/literal` instead. Most modern SOAP frameworks have followed the WS-I recommendations and have abandoned the once famed `RPC/encoded` style in favor of `document/literal`.

A special version of `document/literal` which is frequently used is `document/wrapped`, or `document/literal` with wrapped parameters. `Document/wrapped` is `document/literal` with a few additional confinements:

- Messages in `document/wrapped` may not have more than one message part.
- The type of this single part (the *wrapper*) has a local name equal to the operation name of the operation that this message is associated with.
- The wrapper type is defined using the `sequence` compositor. Other compositors (`all` or `choice`) may not be used.
- The wrapper type has no attributes.

In practice this means that the parameters of `document/wrapped` operations are all wrapped up in a single type, which has a name equal to the name of the operation to invoke. This has a few practical advantages: the operation name is contained in the SOAP message (as the name of the wrapper type), and messages structured like that can be validated against the schema contained in the WSDL description with a standard XML Schema validator. The main disadvantage of the style is that it can not support overloading of WSDL operations [15]. Listing C in the appendix uses the `document/wrapped` WSDL-to-SOAP binding style.

2.4 REST

A different flavor of Web services was inspired by Fielding's *Representational State Transfer* (REST) architectural style [33, 34]. In its most general form (the form originally described by Fielding) REST is a catch-all way of building extremely large-scale distributed systems, and the architectural style that guided the development of the HTTP [32] and URI (Universal Ressource Identifier) [9] standards. Today the synonym "REST" is mostly associated with a certain ("RESTful") way of building services and SOAs.

The motivation behind RESTful services (often seen as opposed to SOAP/WSDL-based services) has been expressed quite well in [65]:

> The problem is, most of today's "web services" have nothing to do with the Web. In opposition to the Web's simplicity, they espouse a heavyweight architecture for distributed object access, similar to COM or CORBA. Today's "web service" architectures reinvent or ignore every feature that makes the Web successful.

As a consequence, the REST community strives to promote Web services that are in line with the ways of the WWW, instead of simply building an entirely new protocol suite on top of the established Web standards HTTP and XML.

RESTful Web services are defined by the following properties[1] [65]:

1. *Resource-orientation* – RESTful Web services are all about data. In the end, most services are all about creating, retrieving or modifying some kind of data. These resources are therefore the central element in a REST architecture. Ressources are often represented in XML, although other representations such as HTML, JSON (JavaScript Object Notation) or plain text are also perfectly valid.

2. *Addressability* – Every resource managed by a Web service has to be directly addressable via an URI. The URI of a resource has to contain everything necessary to uniquely identify the resource.

3. *Statelessness* – RESTful Web services do not maintain a state. The state of a client is entirely managed by the client, or encapsulated in the resources.

4. *Uniform interface* – Basically, every RESTful service exposes the same (uniform) interface: REST allows you to *read* resources (GET), *create* resources (PUT or POST), *modify resources* (PUT) or delete them (DELETE). GET is a *safe* operation (it does not alter the state of any resource), PUT and DELETE are *idempotent* (issuing *n* DELETE or PUT operations has the same effect as doing it just once).

[1]Actually, [65] uses these properties to define a special type of RESTful architecture, the "Resource-Oriented Architecture", but the author of this thesis thinks that these properties are a good definition of any type of RESTful architecture.

Theses properties also mark the main differences between a RESTful service and the way SOAP/WSDL-based services are built. Snell states in an IBM online article from 2004 that the main difference between RESTful and SOAP-based services is that the first are centered on the notion of resources, while the second are based on the notion of "activities" [73]. Property 2 is also a distinct differentiator: SOAP-based services are available through one or a few endpoints; nothing in a SOAP-based service is actually addressable. Property 4 means that SOAP and RESTful services use HTTP in a completely different manner: SOAP-based services use only HTTP POST to transport SOAP messages (basically reducing HTTP to data transmission, ignoring any other capabilities of the protocol). RESTful services on the other hand use the full HTTP standard. They do not encode method information (the "what do you want to do" information) in the request entity, but already in the chosen HTTP method.

2.4.1 Interface Description in REST

```
<application xmlns="http://research.sun.com/wadl"
  xmlns:ex="http://my.example.namespace/">

  <grammars>
    <include href="MyServiceSchema.xsd" />
  </grammars>

  <resources base="http://my.search.com/MySearch/Resource1/">
    <resource uri="mySearch">
      <method href="#search" />
    </resource>
  </resources>

  <method name="GET" id="search">
    <request>
      <param name="searchstring" style="query"
        type="xsd:string" required="true" />
    </request>
    <response>
      <representation mediaType="application/xml"
        element="ex:MySearchResult" />
    </response>
  </method>

</application>
```

Listing 3: WADL example

Property 4 (uniform interface) also means that an interface definition language such as WSDL is not nearly as important for REST as it is for SOAP-based services. As a direct consequence no such language has yet seen wide use. One promising attempt is WADL, the Web Application Description Language [46, 65]. WADL describes

RESTful Web services by defining what resources exist in a service, how they are represented, how they are connected and what HTTP methods can be used on them. XML representations of resources are described in either XML Schema or RelaxNG [37]; equally expressive languages for other formats (e.g. HTTP, JSON, ...) are currently not available.

Listing 3 shows a minimal WADL example. It defines a simple service that can be invoked by issuing a HTTP GET request to `http://my.search.com/MySearch/Resource1?searchstring=WADL` (note the request parameter encoded in the request). The service will return the search results in XML representation (as defined in the XML Schema `MyServiceSchema.xsd`).

Unfortunately WADL is supported very poorly in the Web service community so far. There are only basic frameworks to process WADL available, and almost no RESTful service provides a WADL interface description.

Another (less formal) way of describing the usage of RESTful services is to simply provide *example requests* (for instance on the homepage of the service). This is often done as part of the service documentation. These samples are not intended to be machine-processable, and are often incomplete or fragmented, but in theory a given example can be automatically parsed and used as a “blueprint” for further invocations. During the remainder of this theses this approach to interface definition will be referred to as *REST by example*.

3 Related Work

You don't have to be a "person of influence" to be influential. In fact, the most influential people in my life are probably not even aware of the things they've taught me.
– Scott Adams

This chapter gives an overview over relevant existing work in the field of dynamic and asynchronous invocation of (SOAP-based) Web services. It will show examples of how current SOAP frameworks in Java handle stubless service invocation, and discuss what the limitations of these solutions are. It will also clarify how dynamically typed languages such as Perl handle these issues. Subsequently, existing work in the area of asynchronous service invocation will be presented.

3.1 Dynamic Service Invocation

As described in Chapter 2 the main advantage that Service-Oriented Architectures can add to today's software landscape is an extremely *loose coupling* of services: a service-aware client may dynamically (at run-time, as opposed to design- or compile-time) select an appropriate service that implements the specific functionality that the client needs, connect to it and invoke it over standardized interfaces. If the service does not perform well or becomes unavailable the client might just change to another service implementing a similar functionality. Three steps are fundamental to achieve such a loose coupling of service requesters and providers: providers *publish* service descriptions in a standardized language (usually WSDL, the Web Service Definition Language), requesters *find* services that deliver functionality that they need, and finally requesters *bind* to these services and invoke them. This Section will detail how the last of these steps, dynamically binding to services, can be achieved with current SOAP frameworks.

3.1.1 WSIF

The Web Service Invocation Framework (WSIF) [6] is a client-side Web service framework for the Java programming language that "allows the application-programmer to program against an abstract service description, in a protocol-independent manner" [25]. WSIF also pioneered the idea of dynamic Web service invocation for Java: it was the first major framework that included an explicit dynamic invocation interface. Listing 4 shows the usage of the DII to invoke a simple service.

```
// create WSIF port
WSIFServiceFactory factory = WSIFServiceFactory.newInstance();
WSIFService service = factory.getService(
  "http://my.service.com/wsdl", // path to WSDL
  "http://my.namespace.com",    // service namespace
  "MyService",                  // service name
  "http://my.namespace.com",    // port type namespace
  "MyPort");                    // port type name
WSIFPort port = service.getPort();

// create operations and messages
WSIFOperation operation = port.createOperation("helloWorld");
WSIFMessage input = operation.createInputMessage();
WSIFMessage output = operation.createOutputMessage();
WSIFMessage fault = operation.createFaultMessage();
input.setObjectPart("helloWorld", "Hello WSIF!");

// fire invocation
operation.executeRequestResponseOperation(input, output, fault);
String response = (String) output.getObjectPart("return");
```

Listing 4: Dynamic invocation in WSIF

Dynamic invocation in WSIF as displayed in Listing 4 is very intuitive, but there is one caveat: in principle the WSIF DII can handle only a limited number of predefined data types (such as string, integer, short, ...). Whenever a user-defined type should be transferred over a service the type has to be "mapped" to a Java type before starting the invocation. Listing 5 shows an example where an XML type `MyType` is mapped to `MyJavaType`.

```
service.mapType(new QName("http://my.namespace.com", "MyType"),
  MyJavaType.class);
```

Listing 5: Type mapping in WSIF

The downside is that mapping types as in Listing 5 can only be done if there is a matching Java type readily available. In a truly dynamic invocation scenario (where no type information is available at design-time) the WSIF DII can hardly be used.

Today the WSIF project is loosing ground to newer Web service frameworks. The latest version is already from 2003, and there is no active work on the code base at the moment. WSIF therefore does not support the newer trends in Web services, for instance REST, MTOM (SOAP Message Transmission Optimization Mechanism) [88] or most of the WS-* stack. WSIF is also outdated in terms of runtime performance by now (cp. Chapter 5).

3.1.2 Apache Axis 2

Apache Axis 2 [4] is the direct successor of the well-known Apache Axis [3] Web service framework. It is a complete re-write of the original Axis code, utilizes an entirely new architecture and is claimed to be "more efficient, more modular and more XML-oriented than the older version" [4]. One of the cornerstones of Axis 2 is AXIOM (AXis Object Model) [2], the XML object model built within the scope of the Axis 2 project. AXIOM is an efficient XML model based on XML Pull Parsing [72] (more concretely on StAX, the Streaming API for XML [20]).

Apache Axis 2 incorporates a lot more of the SOA concepts described in Chapter 2 than its predecessor: it supports client-side asynchrony (with explicit support for SOAP/WSDL Message Exchange Patterns) and works much more on a document level than the strictly RPC-based Apache Axis. Axis 2 is still grounded on the usage of client-side stubs to implement Web service clients, but it also supports dynamic invocations through two slightly different APIs: the `OperationClient` API provides full access to the internal DII of Axis 2, the `ServiceClient` API is an abstraction of `OperationClient` and more straightforward to use. Listing 6 shows an DII example of Axis 2 using the `ServiceClient` interface.

```
1  // create and configure the Service Client
2  ServiceClient sender = new ServiceClient();
3  Options axis2Options = new Options();
4  axis2Options.setTo(new EndpointReference(
5    "http://my.service.com/epr"));
6  sender.setOptions(axis2Options);
7
8  // create SOAP payload
9  OMFactory fac = OMAbstractFactory.getOMFactory();
10 OMNamespace ns = fac.createOMNamespace(
11   "http://my.namespace.com", ex);
12 OMElement request = fac.createOMElement("helloWorld", ns);
13 request.addChild(fac.createOMText(helloWorld", "Hello World!"));
14
15 // fire invocation
16 OMElement result = sender.sendReceive(request);
```

Listing 6: Dynamic invocation in Axis 2

A closer examination of Listing 6 discloses the big drawback of the Axis 2 DII: step 2 in the listing (lines 8 to 13) requires the application developer to manually construct the entire XML payload of the SOAP message (as AXIOM model). In this case Axis 2 only takes care of the network transmission, the actual SOAP encoding of the invocation has to be handled by the client-side application. The `OperationClient` API is not better in this context; it allows the application developer to set more details of the Axis 2 invoker (for instance the `SOAPAction`), but the SOAP payload still has to be constructed on application side.

Apache Axis 2 definitely is a step into the right direction (as compared to the first version), but the DII still does not provide the support that would be necessary to implement real-world SOA scenarios. In order to use Axis 2 in such scenarios a wrapper has to be implemented that encapsulates the complexities of SOAP payload encoding in accordance with a given WSDL definition. The Daios framework as described in Chapter 4 will provide such a wrapper. Daios internally uses the AXIOM XML model for XML processing, and it is possible to use the Axis 2 SOAP stack for data transmission (as one of two alternatives).

3.1.3 JAX-WS

JAX-WS (Java API for XML-based Web Services) is the "official" specification of how Web services in Java should be handled. JAX-WS is described in JSR (Java Specification Request) 224 [50], and is the official follow-up to JAX-RPC [44]. The specification has been renamed by reason that one of the distinct goals of JAX-WS is to foster a document-centric approach (as opposed to the strictly RPC-based JAX-RPC). JAX-WS includes support for dynamic invocation and client-side asynchrony. Similarly to most other recent Web services frameworks it does not support `RPC/encoded` WSDL encoding any longer and focuses on `document/wrapped` instead. JAX-WS states to include support for all XML-based Web services (hence including RESTful services), but clearly the specification is only focussing on SOAP/WSDL-based services.

JAX-WS provides two different APIs for dynamic service invocation: the `Proxy` API is using the facilities already provided by the Java programming language (`java.lang.reflect.Proxy`). This interface can be used to create an invocation interface to an existing *Service Endpoint Interface*. Listing 7 exemplifies this. Note that this invocation does not use a statically generated stub, but still needs an interface corresponding (after the rules defined in the JAX-WS specification [50]) to the WSDL description of the service.

Obviously the reliance on an existing Service Endpoint Interface limits the usefulness of the `Proxy` interface for truly dynamic invocations.

The other DII that JAX-WS specifies is the lower-level `Dispatch` interface. It pro-

```java
Service service = Service.create(
  new URL("http://my.service.com/wsdl"),
  new QName("http://my.service.com","MyService"));

MyExampleInterface sei = service.getPort("MyService",
  MyExampleInterface.class);

sei.helloWorld("Hello World!");
```

Listing 7: JAX-WS Proxy example

vides methods that allow the application developer to work on XML message level. This interface supports genuinely dynamic invocations, but (much like the Apache Axis 2 DII described in Section 3.1.2) demands for in-depth knowledge of SOAP and the XML structure of Web service messages to be of any use.

Both the `Proxy` and the `Dispatch` API include client-side asynchrony. They support the asynchrony patterns `Fire and Forget`, `Poll Object` as well as `Callback` (see Section 3.2.1).

3.1.4 Other Java Frameworks

There are plenty of other Java-based frameworks for SOAP/WSDL Web services. Specifically important for this thesis are (besides Apache Axis 2 and Apache WSIF) Codehaus XFire [21] and Apache CXF [5]. CXF is a relatively new contestant on the hard-fought Web services market, and the continuation project of the well-established XFire. CXF is the result of a merger of the XFire project with the Celtix [58] Java ESB runtime. Axis 2, WSIF, XFire and CXF will be evaluated in Chapter 5 of this thesis.

3.1.5 SOAP::Lite

Most of the problems with dynamic service invocation in languages such as Java come from the strong emphasis on providing stubs and trying to achieve *static type safety* when invoking services. Dynamically typed languages such as Perl, Python or Ruby do not support the concept of pre-compiled stubs or static type safety. Dynamic service invocation is therefore often extremely simple in such languages.

Listing 8 clarifies this fact: it shows the dynamic invocation of a Web service using Perl and the SOAP::Lite [74] module. SOAP::Lite “is a collection of Perl modules which provides a simple and lightweight interface to the Simple Object Access Pro-

```
my $soap = SOAP::Lite
     -> proxy('http://my.service.com/epr')
     -> helloWorld(SOAP::Data->name(in0=>'Hello World!'));

my $result = $soap -> result();
```

Listing 8: SOAP invocation in Perl using SOAP::Lite

tocol (...) both on client and server side." [74]. SOAP::Lite is a well-established SOAP framework that is very easy to use and extremely stable. On the other hand it is slowly losing shares for its traditional emphasis on `RPC/encoded` SOAP style.

Listing 8 is remarkably concise: basically one line of Perl code suffices to do the Web service invocation. SOAP::Lite really reduces dynamic Web service invocation to the absolutely necessary (endpoint address, operation name and parameters). However, SOAP::Lite is obviously not message-oriented, and there is no support for client-side asynchrony or anything besides standard SOAP (that means no REST support, no MTOM, not even good support for `document/literal` or `document/wrapped` encoding).

3.1.6 Dynamic Invocation using XML Type Subsumption

Within the scope of this thesis the concept is followed that dynamic service invocation can only be achieved by doing entirely without static stubs. Nagano et al. [57] have presented a slightly different approach to the problem. They propose to continue using stubs, but to bind them to "functional" interfaces instead of "precise" ones, so that the same stub is able to invoke any service with a *similar* signature. This approach allows for static type safety while retaining the possibility to exchange service providers at run-time.

The main research problem here is how similarity of interfaces is defined. [57] uses the concept of *XML type subsumption* [52] to define structural similarity in XML Schema types. If two different (but similar) WSDL definitions are compared four different types of similarity can be observed:

1. Element names are identical. Example: both definitions contain an element "name".
2. Element names are not identical, but synonymous. Example: one definition contains an element "name", the other contains an element "label". Note that

it is usually domain-dependent whether two element names can be considered synonymous. Name and label may be synonymous if they refer to product names, but not if they refer to the names of persons (since label is usually not used for persons). Detecting synonymous element names is therefore a hard and most of all domain-dependent problem.

3. Two or more element names from one definition are merged in the other one. Example: one definition contains the elements “name” and “address”, and the other contains a complex element “contact” with the subelements “name” and “address’.

4. Elements are only present in the one or the other definition.

For the cases 1-3 it is usually very easy to convert a message that is compliant to one of the definitions to the other format. For case 4 conversion is also possible, but assumes that the concerning element is not mandatory.

The idea explored in [57] is in theory very promising: it allows to combine the undoubted gains of static type safety and the simpler programming model of static stubs with the ability to exchange service providers at run-time. Unfortunately there are a few practical issues that have to be taken into account: XML type subsumption obviously only compares interfaces at a syntactical level; there is no way how such an algorithm can estimate whether two elements with the same name are actually having the same semantics. Case 2 demands for a well-developed catalog of synonyms for the problem domain in question. Additionally the concept is only feasible for Web services which are defined in an XML-based, formalized interface definition language (what is currently not usual for RESTful services). Therefore the concept was not adopted for the Daios system, but the idea of interface similarity has been reused in a slightly different context within the Daios framework.

3.1.7 Design Patterns for stubless Service Invocation

Patterns are tested solutions to recurring problems in Software Engineering, which are often accepted as the “best” solution to the problem in question. Patterns are rarely really ingenious or innovative, but they often give names to concepts that programmers have been using intuitively at all times. One additional benefit of patterns is that they are often composable: programmers may combine several single patterns to construct more complex structures, sometimes referred to as composite patterns [66] or compound patterns [82] [2]. Software Engineering patterns exist at several levels of abstraction: the famous "Gang of Four" (GoF) has mostly published *design patterns* [41], i.e. patterns at a fine-grained level of abstraction, often implemented using only a couple of classes; the POSA series on the other hand describes *architectural patterns* [13, 14, 69], i.e. patterns at architectural level.

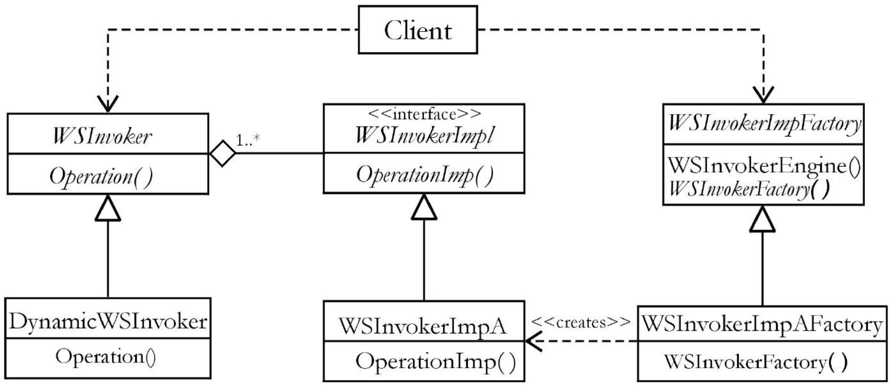


Figure 6: CPWSI in UML notion, taken from [12]

Buhler et al. proposed a compound pattern for Web service invocation [12]. Their *Composite Pattern for stubless Web service Invocation* (CPWSI) aims at separating the application interface from the implementation of the service invocation. CPWSI combines two GoF design patterns, `Factory Method` and `Bridge` [41]. `Factory Method` is a creational pattern which is often used to create objects of (at design-time) unknown type. `Bridge` separates an abstraction (in the concrete case the interface towards the client application) from its implementation (the actual Web service backend). Figure 6 shows CPWSI in UML (Unified Modelling Language) [68] notion.

CPWSI "provides an agile software design for the decoupling of the invocation services that are found and bound at run-time" [12]. The pattern is therefore an excellent design choice for dynamic Web service and SOA interfaces. The fundamental design of the Daios framework as described in Chapter 4 has been vigorously influenced by CPWSI.

3.1.8 Other Approaches

The problems with dynamic service invocation have given rise to a number of "hacks" and workarounds: one often seen "best practice" is described for example in [51]: they

[2] In the meantime the community seems to have settled on the term "compound pattern" since it is not so easily mistaken for the GoF `Composite` design pattern [13, 41].

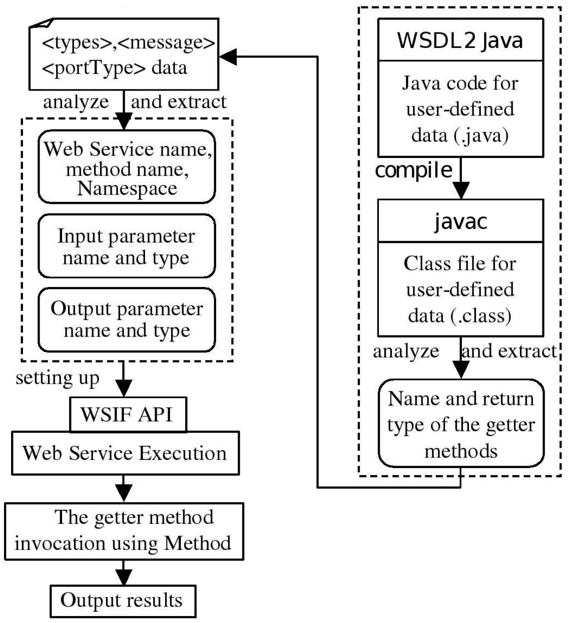

<types>,<message>
<portType> data
analyze and extract
Web Service name, method name, Namespace
Input parameter name and type
Output parameter name and type
setting up
WSIF API
Web Service Execution
The getter method invocation using Method
Output results
WSDL2 Java
Java code for user-defined data (.java)
compile
javac
Class file for user-defined data (.class)
analyze and extract
Name and return type of the getter methods

Figure 7: Service invocation using reflection, simplified from [51]

use Axis' `WSDL2Java` tool to generate Java types for types from WSDL definitions. These are then compiled using the standard Java compiler `javac`, loaded into the classpath and analyzed by means of the Java reflection API. Then they use the WSIF DII to map the newly compiled Java types to the original WSDL types (as in Listing 5) and invoke the service (as in Listing 4). Finally they extract and analyze the Web service result (again using reflection). The overall procedure is sketched in figure 7.

It is obvious that this practice can only be seen as a temporary solution: generating bytecode at runtime is very expensive in terms of performance, and programming with introspection (reflection) is extremely error-prone. Still, lacking a superior alternative, practitioners are forced to employ solutions similar to the one described on a regular basis.

3.2 Message-based and Asynchronous Service Invocation

Today's Web services are predominantly used in an RPC fashion instead of message- and document-oriented. This is fundamentally against the idea of SOAs and SoC and induces some disadvantages: RPC usually implies simple request/response communication, i.e. the communication between client and server is *synchronous*; the client thread blocks until the server delivers a result. Even `void` invocations (invocations without result) are generally executed synchronously, thus blocking the client unnecessarily until the server sends a "finished" signal. Message-based communication is in contrast usually carried out in an *asynchronous* fashion: client and server are decoupled; the client does not block until he receives a result from the server. What is even more important is that message-based communication also helps to decouple clients and server. Message-based systems do not imply as severe interface dependencies as RPC-based communication.

3.2.1 Client-side Asynchrony Patterns

Asynchronous communication is multifaceted; there are various subtly different flavors of asynchronous interactions. In [82], four different patterns of client-side asynchrony are described:

- `Fire and Forget` is a "best-effort" communication pattern; the client tries to deliver the invocation to the server, but successful delivery is not ensured. Errors and failures are not reported to the client. `Fire and Forget` does not support return messages.
- `Sync with Server` extends `Fire and Forget` with delivery confirmations; the client delivers the invocation to the server, and makes sure that the invocation is received successfully. Return messages are not supported. Errors and failures are only reported if they affect the delivery of the invocation.
- `Poll Objects` are full-fledged asynchronous invocations which deliver invocations reliably and allow for return messages. Returned data is (as soon as it becomes available) stored in a special stub, the poll object. Clients can check ("poll") the stub when it is convenient for them to retrieve invocation results.
- `Result Callback` can be used as an alternative to `Poll Objects`. Just like the former it supports return messages, but they are delivered differently. When using `Result Callback` the client registers a specific callback handler which is notified as soon as the result is available. The client therefore does not need to poll for the result, and is interrupted on arrival of the invocation result.

Table 2 compares the four variants of client-side asynchrony. `Poll Objects` and `Result Callback` are obviously the most powerful patterns, but they also make some demands on the framework and the client-side developer. `Fire and Forget` or `Sync with Server` should be used when no result message is necessary (since these patterns are simpler to handle and less ressource-intensive).

Pattern	Reliable Delivery	Result Message	Result Delivery
`Fire and Forget`	No	Not supported	n/a
`Sync with Server`	Yes	Not supported	n/a
`Poll Objects`	Yes	Supported	Client has to "poll" for the result (pull approach)
`Result Callback`	Yes	Supported	Client is informed via call-back (push approach)

Table 2: Client-side asynchrony patterns, after [82]

SAIWS (Simple Asynchronous Invocation Framework for Web Services) [71] is an asynchronous Web service invocation frontend that is built around the client-side asynchrony patterns described earlier [94, 95]. SAIWS supports all four asynchrony patterns from above. It is implemented as a requester [82], and uses Apache Axis as SOAP backend.

3.2.2 SSDL

Some work on asynchronous and message-based Web service invocation has been carried out as part of the SSDL project. SSDL (SOAP Service Description Language) is "designed for describing asynchronous, message-oriented, and multi-message interactions between Web services" [63]. SSDL utilizes existing technologies such as XML, SOAP and WS-Addressing [86]. The notion of invocations or operations is intentionally disregarded, SSDL is instead based on the concept of *one-way messages*. As in WSDL these one-way messages can be grouped into specific MEPs.

The central elements of SSDL are SSDL contracts: contracts "provide the mechanisms (. . .) to describe the structure of SOAP messages that a Web service supports" [63]. SSDL contracts are therefore similar to WSDL definitions, but with broader scope. They make use of so-called SSDL protocols, i.e. well-defined sequences of messages that the service accepts.

Listing 9 shows how SSDL protocols are defined: the example uses the *Sequencing Constraints* (SC) framework to arrange two messages in a simple request/response style. Appendix D extends the example from Listing 9 to a full SSDL contract: types are defined using XML Schema, these types are used to declare messages, the messages are arranged in communication protocols, and finally the concrete service

```
<!-- define messages -->
<messages>
  <message name="InMessage">
    <header ref="myInType" />
  </message>
  <message name="OutMessage"
    <header ref="myOutType" />
  </message>
</messages>

<!-- define protocols -->
<protocols targetNamespace="urn:service:sc">
  <protocol name="exampleProtocol">
    <sc>
      <sequence>
        <msgref ref="InMessage" direction="in" />
        <msgref ref="OutMessage" direction="out" />
      </sequence>
    </sc>
  </protocol>
<protocols>
```

Listing 9: SSDL protocol definition

endpoints are defined using WS-Addressing.

SSDL is conceptually related to Daios since it also addresses many of the problems that motivated the development of the Daios framework: SSDL criticizes WSDL for it's focus on the notion of operations and the predominance of synchronous RPC style that comes along, and endorses asynchronous and message-based communication instead. However, the scope of SSDL is significantly different to Daios' scope: SSDL concentrates on interactions between Web services, Daios on the other hand works on the actual Service invocation level. SSDL is therefore more of a competitor to Web service choreography languages, and can be regarded as complementary to Daios.

3.2.3 WSMQ

WSMQ (Web Services Message Queue) is a specific message-oriented middleware implementation for Web services [55, 56]. It aims to foster reliability, scalability and security of Web services while at the same time maintaining all original advantages of the technology.

Figure 8 displays the general architecture of WSMQ. At client-side an additional component has to be in place, the *Web service simulation*. This "simulator" is implemented as an Interceptor [14] and redirects any Web service invocations to WSMQ. In the WSMQ system Web service requests are received, classified and put

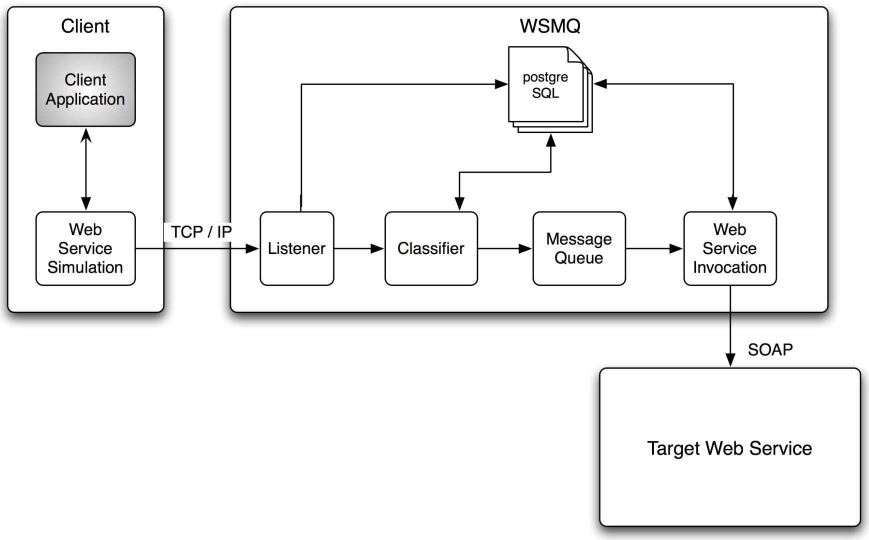


Figure 8: WSMQ overview, after [55]

into queue. Then the invocations are directed to the original target service, the result is received and sent back to the client. To the client the whole process looks identical to a standard invocation without WSMQ intercepting.

This additional level of indirection has a few advantages:

- *Increased reliability* - usually a Web service invocation simply fails if the server becomes unavailable before or during an invocation. WSMQ has the possibility to store the request and retry the invocation at a later time. This makes the Web service invocation more reliable.
- *Decoupling in time* - standard Web service invocations are coupled in time - client and server have to be online at the same time in order to be able to collaborate. A message queue can loosen this requirement since it can store requests for later invocations. This way server and client can be online at different times and still collaborate.
- *Load balancing and performance* - naively it could be suspected that the additional layer of indirection reduces the performance of Web service invocations. According to experiments described in [56] this is not so much the case. Since WSMQ is able to balance concurrent invocations targeted at the same service the overall runtime performance (the time it takes the clients to finish invocations) in their test scenario is effectively better than without WSMQ

intercepting. However, [56] admits that WSMQs performance may vary in real-world scenarios.

WSMQ is a good approach to unifying the advantages of MOM systems and Web service technology. It should however be mentioned that many of the advantages of WSMQ (reliability, decoupling in time, . . .) are also provided by *Enterprise Service Bus* (ESB) technologies [18, 60, 70], which are currently getting a lot more research echo than message queues such as WSMQ. Other authors are trying to use existing messaging technologies for Web services instead of developing new ones (see [76] for an example).

4 Design and Implementation

See first that the design is wise and just;
That ascertained, pursue it resolutely;
Do not for one repulse forego the purpose
That you resolved to effect.
– William Shakespeare, source unknown

Chapter 1 has introduced a set of desirable features for Web service clients that truly support the SOA vision: dynamic service invocation, message-orientation, client-side asynchrony and support for SOAP as well as REST-based services. Chapter 3 has described among other things how current Web service stacks and specifications can deal with these requirements.

The following Chapter will detail the architecture and implementation of the Daios (Dynamic and asynchronous invocation of services) framework, which has been developed with special consideration of the requirements defined in Section 1.2.1. Daios is a Web service invocation frontend for SOAP/WSDL-based and RESTful services. It supports only fully dynamic invocations without any static components such as stubs or Service Endpoint Interfaces.

Daios is grounded on the notion of message exchange: Daios clients communicate with services by passing messages (`DaiosInputMessage`s) to them; services return the invocation result by answering with messages (`DaiosOutputMessage`s). These Daios messages are potent enough to encapsulate XML Schema complex types, but much simpler to use than working directly on XML message level. Daios will take care of converting `DaiosInputMessage`s into Web service invocations (for instance by converting them to SOAP according to a given WSDL definition), issue the invocation, receive the result from the service and convert the result back into a `DaiosOutputMessage`. This procedure abstracts most of the RPC-like internals of SOAP and WSDL; the client-side application does not need to know about WSDL operations, messages, endpoints or encoding. Even whether the target service is implemented as SOAP- or REST-based service is (almost) transparent to the client application.

The backend used to conduct the actual invocation is replaceable: the Daios research prototype comes with two options of *invocation backends*, one which uses the Apache Axis 2 stack and one which utilizes a custom-built ("native") SOAP and

REST stack. Unless stated otherwise the rest of this description will focus on the native backend. Daios also puts much emphasis on client-side asynchrony. All invocations, and using any backend, can be issued either in a blocking way, or as `fire and forget`, `callback` or `poll object` invocations as explained by the client-side asynchrony patterns described in Section 3.2.1.

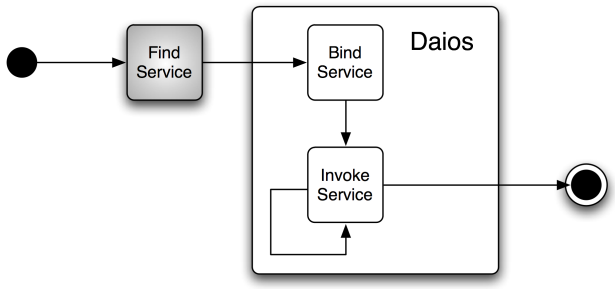


Figure 9: General dynamic invocation procedure

The general procedure of dynamic Web service invocations with the Daios framework is shown in Figure 9. Firstly clients have to find a service that they want to invoke. This step is external to Daios (cp. Chapter 1). Afterwards the service has to be bound. In Daios this step is called the *preprocessing phase*. During the preprocessing Daios will compile the definition of the service into an internal representation. For WSDL-based services this step includes compiling the WSDL definitions file and the XML Schema contained therein, for RESTful services an example invocation for "REST by example" can be loaded. When the service is successfully bound clients can issue any number of invocations to this service until they decide to release the service binding again. Service bindings can be kept alive for an unlimited amount of time – bindings are only abstract references and do not imply a permanent connection to the service, and therefore do not stress the server in any way. However, if the service (i.e. the interface definition of the service) changes the binding has to be renewed in any case. Automatic change detection is currently not supported by the framework and therefore has to be handled by the client application if service bindings are to be kept alive for a long time.

4.1 Architecture

Figure 10 sketches the general architecture of the Daios framework. The framework internally splits up into three functional components: the general Daios classes which are the core of the framework and orchestrate the individual other components, the interface parsing component which is responsible for preprocessing and the invoker component which conducts the actual Web service invocations using a REST or

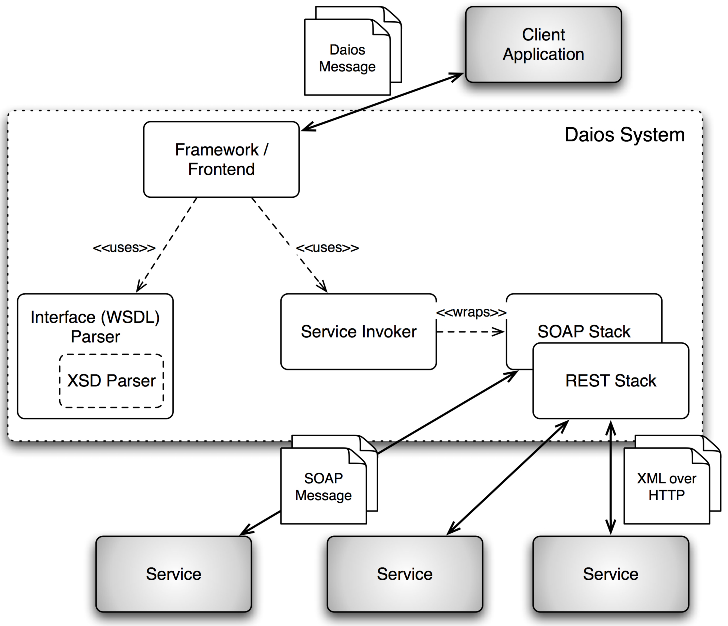


Figure 10: Daios overall architecture

SOAP stack. Clients communicate with the framework frontend using Daios messages which are Daios' internal data representation format.

The core classes that are considered to be part of the general framework are vital to the Daios system and irreplaceable. The interface parsing component is separated from the rest of the system and can be replaced if necessary. The service invoker component can be chosen and replaced during run-time. All of these components will be described in more detail below.

4.1.1 Frontend

The *frontend* component contains everything that is absolutely necessary for the Daios system to work: the interface to the client, the framework that orchestrates the other components and a number of utility classes that provide Daios-specific algorithms and procedures. Figure 11 shows a simplified UML class diagram detailing the internal structure of the Daios framework.

Figure 11 is an instantiation of the CPWSI pattern (see Section 3.1.7 or [12]). The GoF `Bridge` pattern [41] is used to separate the Daios interface from the actual

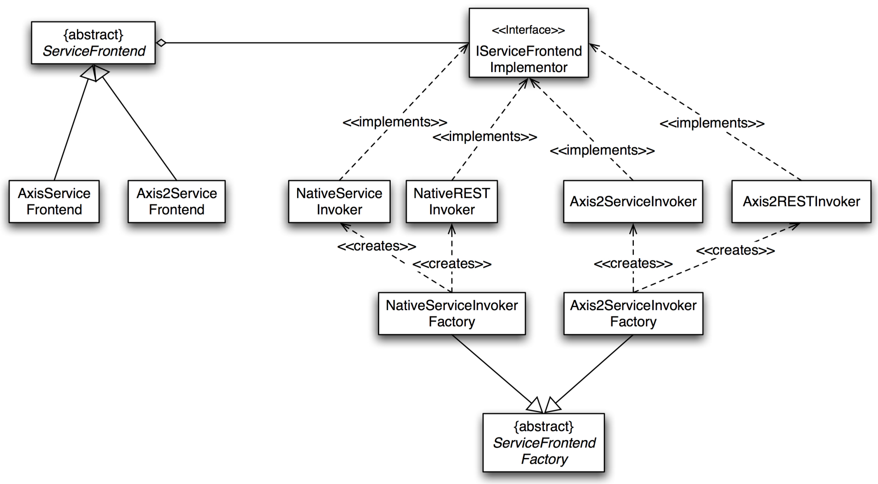


Figure 11: Daios Frontend Structure in UML syntax

backend implementation. The `Factory Method` pattern [41] allows the client to dynamically choose a backend at run-time by passing the fully qualified class name to the factory. Currently Daios includes two backend implementations, the Apache Axis 2 backend and the native backend. Table 3 shows these currently available backends and the associated factory class names. Additional backend implementations can be provided easily; no code change in the framework is necessary for such an adaption. Listing 10 exemplifies the creation of a new *Service Frontend*. Service Frontends are `Client Proxies` [13, 82] that wrap concrete Web services (endpoints).

Backend	**Factory Class Name**
Native Backend	`at.ac.tuwien.infosys.dsg.daiosPlugins.` `nativeInvoker.NativeServiceInvokerFactory`
Axis 2 Backend	`at.ac.tuwien.infosys.dsg.daiosPlugins.` `axis2.Axis2ServiceInvokerFactory`

Table 3: Factory class names of Daios backends

Listing 10 instantiates a new frontend factory (a factory for the native backend), and uses this factory to create a Service Frontend to the SOAP-based service defined in the WSDL file `http://my.service.com/wsdl`. The `createFrontend()` method launches the preprocessing phase. Once these methods have returned a new Service Frontend for the service is bound and ready for invocations.

A selection of other important utility components that are considered to be part of the general framework are listed in Table 4.

```
ServiceFrontendFactory fac =
  ServiceFrontendFactory.getFactory
    ("at.ac.tuwien.infosys.dsg.daiosPlugins."+
     "nativeInvoker.NativeServiceInvokerFactory");

ServiceFrontend frontend =
  fac.createFrontend(
    new URL("http://my.service.com/wsdl")
  );
```

Listing 10: Frontend creation in Daios

Component	**Responsibility**
`AtomicTypesMapper`	Provides Java-to-XSD (and vice versa) type mapping for built-in types.
`Matcher`	Measures similarity of Daios messages and WSDL operations (see Section 4.1.4).
`RESTURLEncoder`	Serializes Daios messages as HTTP GET request parameters.
`TypeParser`	Converts an XML Schema type system into an in-memory representation (see Section 4.2.1).
`WSDLTypeTree`	Represents a compiled type system as generated by the `TypeParser`.

Table 4: Utility components in the Daios framework

4.1.2 Daios Messages

Daios messages are the standard data exchange format in the framework. They are used as input to and output from invocations, and encode arrays and complex types. Daios messages are sufficiently powerful to handle arrays, complex XML Schema types and arrays of such, but are relatively simple to use for the client-side developer.

Messages are simply unordered lists of *name-value pairs*, so-called message fields. Every field has an unique name, a type and a value. Valid types are either built-in types (*simple field*), arrays of built-in types (*array field*), complex types (*complex field*) or arrays of complex types (*complex array field*). Table 5 enumerates all available built-in types and their mappings to Java types.

Complex types (types which are not represented by one of the types from Table 5) can be constructed by nesting messages. This way users can construct arbitrarily complex data structures. `java.lang.Object` types are serialized using the `toString()`

XML Schema Type	Java Type
`xsd:int`	`java.lang.Integer`
`xsd:short`	`java.lang.Short`
`xsd:long`	`java.lang.Long`
`xsd:boolean`	`java.lang.Boolean`
`xsd:float`	`java.lang.Float`
`xsd:double`	`java.lang.Double`
`xsd:dateTime`	`java.util.Date`
`xsd:base64Binary`	`java.lang.Byte[]`
`xsd:anyType`	`java.lang.Object`

Table 5: Atomic type mapping in Daios

method of the object, and deserialized using a constructor which takes an XML string as argument. This allows the application developer to write custom serializers and deserializers for `xsd:anyType` placeholders.

Listing 11 exemplifies how Daios messages are constructed. The message has four fields, “name”, “age”, “friends” and “address”. The fields “name” and “age” are simple. The field “address” is a complex field and therefore represented by a nested message, which again contains three simple fields. The “friends” field is an array field and contains an array of simple types. The type definition section of a service description accepting such a message is exemplified in Appendix G.

```
DaiosMessage message = new DaiosMessage();
message.setString("name", "Philipp Leitner");
message.setInt("age", 24);
message.setStringArray("friends",
  new String[] {
    "Sepp Maier",
    "Fritz Huber",
    "Ferdinand Lang"
  });

DaiosMessage address = new DaiosMessage();
address.setString("city", "Vienna");
address.setInt("house",14);
address.setInt("door",15);

message.setComplex("address", address);
```

Listing 11: Constructing Daios messages

4.1.3 Interface Processing

The interface processing component is responsible for the preprocessing as shown in Figure 9. It takes an interface description (in XML notion) as input and provides a parsed in-memory representation of the given description as output. What exactly gets generated depends on the type of interface definition provided: if a WSDL definition is given then the interface processing component will compile it, read the WSDL encoding type, and extract the operation and signature information of all services as well as the according endpoint addresses. Afterwards the XML Schema type system contained in the definition is parsed.

In case of a SOAP invocation a WSDL interface is mandatory; SOAP services which do not provide a formalized WSDL interface are not supported by Daios. In case of a RESTful service an "example invocation" in XML representation may be provided. In that case the interface processing component will parse the example request and store it for later usage. RESTful services may also be used without any formal interface definition; then Daios will simply try to default to the encoding most often seen in RESTful services, simple HTTP GET with URL-encoded request parameters. Other data encoding formats such as JSON are currently not supported.

4.1.4 Service Invoker

The service invocation component is probably the most interesting part of the Daios framework: it can use the parsed information provided by the interface processing component during the preprocessing phase to construct dynamic invocations to SOAP- or REST-based Web services. The invoker component is implemented in an extensible way: new *backends* (i.e. new service invoker implementations) can be introduced into the framework very easily. The only requirement for new backends is that they are derived from the `at.ac.tuwien.infosys.dsg.daios.framework.ServiceFrontend` base class and instantiated using a factory class that inherits from `at.ac.tuwien.infosys.dsg.daios.framework.ServiceFrontendFactory`. A full backend provides two different Web service stacks, one for SOAP- and one for REST-based services. Naturally the course of action during the dynamic invocation is different for both protocols.

SOAP-based Services. Figure 12 pictures the course of action for a SOAP-based invocation.

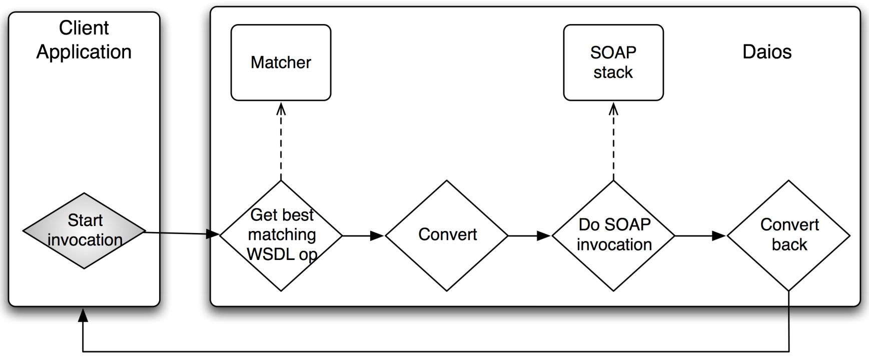


Figure 12: Dynamic SOAP invocation

1. The invocation is started by the client application. It sends a `DaiosInput Message` to the service invoker. In the standard case this message contains everything that the client needs to know about the service. All other relevant invocation metadata (e.g., endpoint address, encoding, ...) is handled by Daios.

2. The first step for the invoker is now to compare the received `DaiosInput Message` to the WSDL operations defined in the service. The invoker therefore uses the facilities provided by the `Matcher` utility component to find a WSDL input message with the lowest *structural distance* to the received Daios message.

3. When the most fitting WSDL input message is found the `DaiosInputMessage` is converted to whatever format the used SOAP stack expects. For the Apache Axis 2 SOAP stack this results in directly converting the `DaiosInputMessage` to SOAP in AXIOM notion (cp. the description of the Axis 2 DII in Section 3.1.2).

4. Then the converted message is passed (along with other necessary invocation information such as the endpoint address to use) to the SOAP stack, which will then carry out the invocation and receive the result.

5. The result is converted back into the Daios-internal representation (i.e. into a `DaiosOutput Message`).

6. Finally the `DaiosOutputMessage` is returned back to the client.

This course of action is similar for synchronous and asynchronous invocations: the main difference is just whether the client is blocked while Daios is proceeding, and how the invocation result (if any) is returned to the client.

The most challenging part of the sequence above is step 2: the `Matcher` implements a *structural distance calculation algorithm* that defines the structural distance of a given Daios message and a message from a WSDL definition. Structural distance calculation follows some of the ideas of [57], and is defined by the following rules:

1. If the WSDL message is no input message (i.e. not used as *input* in any WSDL operation in the definitions) the distance is ∞.
2. If the Daios message contains a field that has no corresponding *part* in the WSDL message the distance is ∞. Daios fields and WSDL parts are considered to be corresponding iff the field name of the Daios field is equal to the part name.
3. If the WSDL part corresponding to a message field is simple and has a different type the distance is ∞. If the message field is complex calculate the distance of the nested Daios message and the type of the WSDL part.
4. Otherwise the distance is the number of WSDL parts without corresponding Daios fields *plus* the sum of all distances of nested complex fields.

This algorithm is sketched as pseudo-code in Appendix E. Note that structural distance is not symmetric: $dist(A, B)$ is not necessarily the same as $dist(B, A)$. The reason is that WSDL messages may have optional parts, i.e. parts that do not have to be specified by the client application, but Daios messages should not contain any information not expected by the service. That means that it is perfectly valid to have WSDL parts not contained in the Daios message (but the structural distance will increase), but as soon as there is one field in the Daios message that cannot be mapped to a WSDL part the distance becomes ∞ and the messages are considered to be totally incompatible.

Daios will choose to invoke the operation whose input message has the lowest structural distance to the provided Daios message. In case no input message has a distance lower than ∞ an error is thrown: the provided input is not compatible with the Web service at all. If two or more messages are bound for the lowest structural distance the client application has to specify the input message to use.

Figure 13 gives an example of the distance calculation: the Daios message depicted left in the figure and the WSDL message in `RPC/encoded` style at the right side are a perfect match, i.e. they have a structural distance of 0. If for instance the field “First Name” would be removed from the Daios message the messages would still be compatible, but have an increased structural distance of 1. If the Daios

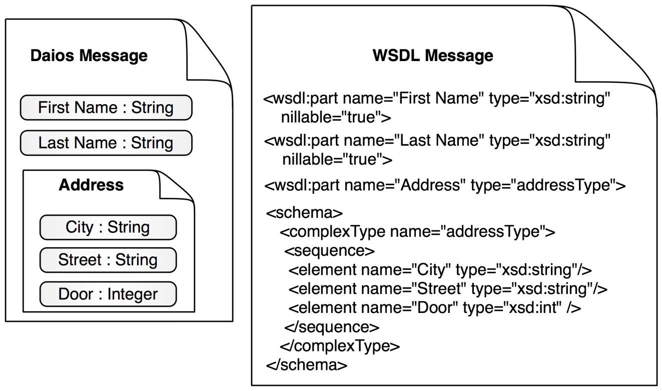

Daios Message
First Name : String
Last Name : String
Address
City : String
Street : String
Door : Integer
WSDL Message
<wsdl:part name="First Name" type="xsd:string" nillable="true">
<wsdl:part name="Last Name" type="xsd:string" nillable="true">
<wsdl:part name="Address" type="addressType">
<schema>
<complexType name="addressType">
<sequence>
<element name="City" type="xsd:string"/>
<element name="Street" type="xsd:string"/>
<element name="Door" type="xsd:int" />
</sequence>
</complexType>
</schema>

Figure 13: Structural distance example

message would be left unchanged, but the part “First Name” removed from the WSDL message, then the messages would be considered incompatible and have a structural distance of ∞. The distance would also be ∞ if the field “Door” in the Daios message would e.g., be of type `Long` instead of `Integer`.

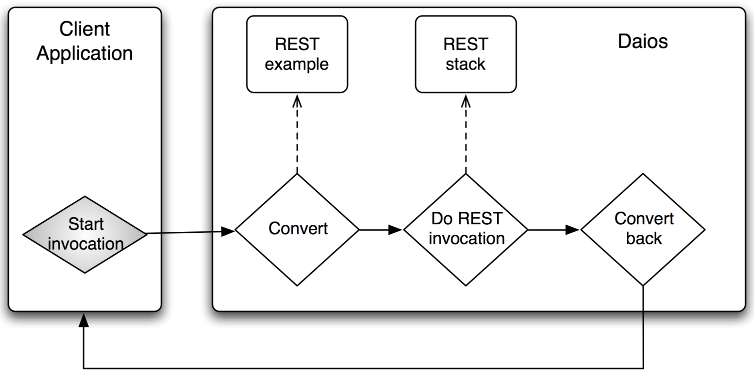
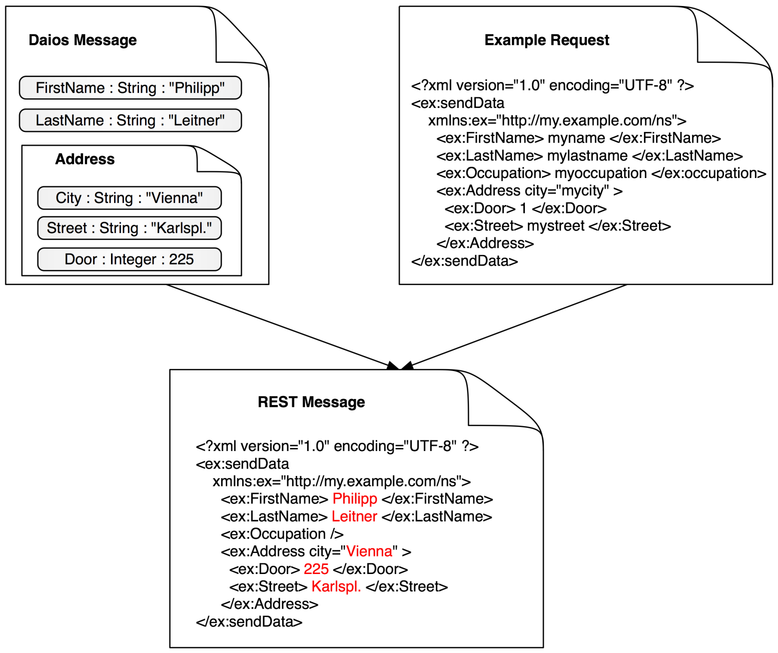


Figure 14: Dynamic REST invocation

REST-based Services. Figure 14 shows the necessary activities for REST invocations. The general sequence is similar to SOAP-based invocations, but for RESTful invocations no distance calculation is needed. Instead the `DaiosInputMessage` is converted to the stack-specific format (either XML or URL-encoded) according to a given example. If no example is given the Daios message is converted to the default URL-encoded form.

The algorithm employed for “REST by example” is currently relatively simple, and is displayed in Appendix F. The algorithm uses the given example as blueprint, and fills all values from the Daios message into elements or attributes with the same name in the example. If at least one field from the Daios message misses a corresponding structure in the example then an error is thrown: the example and the provided input are not compatible. Complex fields are filled recursively. Array fields are represented by a sequence of two or more elements with the same name in the example.



```
<?xml version="1.0" encoding="UTF-8" ?>
<ex:sendData
   xmlns:ex="http://my.example.com/ns">
     <ex:FirstName> myname </ex:FirstName>
     <ex:LastName> mylastname </ex:LastName>
     <ex:Occupation> myoccupation </ex:occupation>
     <ex:Address city="mycity" >
       <ex:Door> 1 </ex:Door>
       <ex:Street> mystreet </ex:Street>
     </ex:Address>
</ex:sendData>
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<ex:sendData
   xmlns:ex="http://my.example.com/ns">
     <ex:FirstName> Philipp </ex:FirstName>
     <ex:LastName> Leitner </ex:LastName>
     <ex:Occupation />
     <ex:Address city="Vienna" >
       <ex:Door> 225 </ex:Door>
       <ex:Street> Karlspl. </ex:Street>
     </ex:Address>
</ex:sendData>
```

Figure 15: REST by example

Figure 15 exemplifies REST by example as employed by Daios: the structure of the example request is filled with the values from the Daios message to produce a REST invocation in XML notion. Note that both XML Elements and Attributes are represented by fields in Daios messages, but attributes are only allowed to by represented by simple fields. Elements can be represented by simple or complex, and as array or non-array types in the Daios message. If an attribute is represented by a complex or array field an error is thrown to indicate that the conversion is not possible.

4.2 Implementation

Daios has been implemented solely using the Java programming language, standard version 6 (Java SE6). The framework is also usable with Java SE5, but the overall performance of the system might be lower. Third-party libraries and code was used whenever that made sense in order to keep the implementation effort low and the quality of the software high. Table 6 enumerates all used third-party software. All of these libraries are published under an open source license.

Library	**Version**
Apache AXIOM	1.2
Apache Axis 2	1.2
Apache Commons ArrayIterator	3.1
Apache Commons HTTPClient	3
Apache XMLBeans	2.2
Codehaus Jaxen	1.1
Codehaus Woodstox	1
soapUI	1.7
WSDL4J	1.6

Table 6: Third-party software used in Daios

One of the most important libraries used is Apache AXIOM. It has been used for all internal XML parsing and representation. AXIOM internally uses the Woodstox implementation of StAX. Apache Axis 2 is used in the Axis 2 backend as SOAP and REST stack. The native backend utilizes HTTPClient for all HTTP communication issues. Jaxen is necessary for evaluating XPath expressions against AXIOM models. WSDL4J is used by the interface processing component to retrieve and parse WSDL definitions. The ArrayIterator utility from the Apache Commons collection is useful for all kinds of reflective programming with arrays in Java. The Web service test tool *soapUI* [31] is not actually a library, but a standalone desktop application. The code used to parse XML Schema type systems contained in WSDL definitions has been extracted from the source code of the soapUI project, and produces an XMLBeans type system.

4.2.1 Implementation Issues

During the development of the Daios research prototype a few previously unexpected implementation issues have become apparent. The hardest problems have been the consistent introduction of XML stream parsing throughout the entire Daios processing chain, as well as the mapping of complex XML Schema types to Java types.

XML Stream Parsing. XML stream parsing increases the performance of XML processing dramatically: if used correctly the XML document is never kept in memory as a whole, instead it is only read and wrote on demand. However, to realize this advantage it is essential that the XML stream is never converted into an in-memory representation at any point during the processing. If e.g., one component converts the stream into a `org.w3c.dom.Document` object all performance gains through stream parsing are lost immediately. That means that application developers need to make sure that no part in the application is buffering and converting the stream. Unfortunately this will often happen implicitly, as side effect of third-party library usage.

Daios uses XML stream parsing through the AXIOM [2] XML object model. In AXIOM the XML model can be seen as a collection of "containers", where every container is representing a part of the XML tree. Each of these containers is either *sourced*, i.e. connected to a input data stream that contains the actual element content, or already consumed, i.e. filled with already known content. As soon as a sourced container is read it becomes consumed: if the content of a sourced element is read for the first time the content is stored in the container and the stream source is discarded.

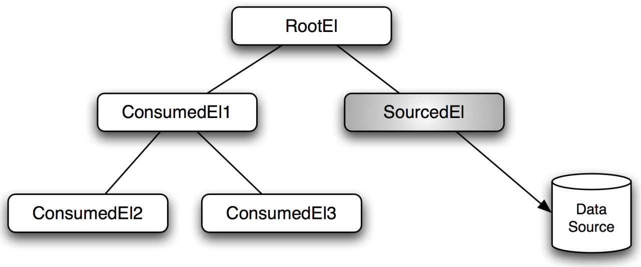


Figure 16: AXIOM XML model

Figure 16 exemplifies this: in the figure the elements `ConsumedEl1` to `ConsumedEl3` are already processed and available in memory. The element `SourcedEl` on the other hand is not consumed yet - the content of this element has yet to be read from the connected data stream. When using AXIOM the developer has to be aware of this structure, as well as of the fact that the full power of AXIOM can only be utilized if sourced elements are never consumed preliminary. Specifically dangerous in that context are the `toString()` and `toStringWithConsume()` methods: both are buffering and reading the whole XML tree, and negating all gains that would come from stream processing if used properly.

In order to use AXIOM in Daios it has been necessary to define Daios-specific data sources. Appendix H shows the input stream data source which has been used to create a fully sourced representation of the return messages in the native backend. Data source implementations have to provide at least a few methods to serialize their content to various types of output streams (`serialize()`), and a method to create a new XML reader that reads the data source content (`getReader()`).

Type Handling. WSDL definitions contain XML Schema type systems to define the data structures used in WSDL parts and messages (cp. Section 2.3.2 or [84, 89]). For the Daios framework it has been a very important issue to parse this type system and create in-memory representations of the WSDL types.

Even though this problem is practically as old as WSDL there is no good off-the-shelf solution available up to now: the WSIF project contains a type parser specifically built for WSDL (`org.apache.wsif.schema.Parser`), but this parser is unable to deal with `sequence` constructs and is therefore basically useless for `document/literal` or `document/wrapped`. XML Schema processors like XMLBeans [7] or Castor [17] are able to process any valid type system, but need to be fed a complete XML Schema document, including all namespace declarations and cross-references. Schemata as contained in WSDL definitions are on the other hand often fragmented into a number of smaller partial schemata, and use namespace declarations from the WSDL parent document. The XML Schema part of a WSDL definition therefore needs additional preprocessing before it can be handed to a XML Schema processor.

```
String style = ... // style contains the WSDL encoding style, e.g.,
                   // document/literal
IWSDL wsdl = ... // wsdl contains the WSDL file to parse in
                 // Daios notion
URL url = ... // url is the original location of the WSDL definitions,
              // necessary to resolve relative imports

WSDLTypeTree types =
  TypeParser.getInstance(style).
  getElementTypes(wsdl, url, style);
```

Listing 12: Parsing types in Daios

For the Daios prototype code from the open source project soapUI [31] has been used. The code fragments extracted from soapUI are contained in the package `at.ac.tuwien.infosys.dsg.daios.wsdl.parser.soapui`, and construct a XML-Beans type system from a WSDL definition. Listing 12 shows the code necessary to kick off type parsing in Daios.

When type parsing is started as in Listing 12 then the soapUI code will extract the XML Schema parts of the WSDL definitions, resolve all external references, add a few necessary standard namespaces, add all used namespaces from the main WSDL file, and feed the resulting array of interdependent XML schemata to the XMLBeans generator, which will produce a Java type system from them. This process is sketched in Listing 13 (the listing is somewhat simplified for brevity).

```
List<XmlObject> schemas = ... // schemas contains the extracted
                              // XML schemata

// create compilation options
XmlOptions options = new XmlOptions();
options.setCompileNoValidation();
options.setCompileNoPvrRule();
options.setValidateTreatLaxAsSkip();
// ... a few more options

// handle namespaces
schemas.add( soapVersion.getSoapEncodingSchema()); // soapenc:
schemas.add( soapVersion.getSoapEnvelopeSchema()); // soapenv:
schemas.addAll( defaultSchemas.values() ); // other necessary schemata

// create type system
SchemaTypeSystem sts = XmlBeans.compileXsd(
  schemas.toArray(new XmlObject[schemas.size()]),
  XmlBeans.getBuiltinTypeSystem(),
  options
);
```

Listing 13: Using XMLBeans to parse WSDL

4.3 Client-Side Interface

The general procedure of using Daios follows the steps displayed in Figure 9. First of all a fitting Web service has to be discovered. This step is not supported by Daios (since the service discovery problem is mostly a registry issue). Afterwards the service has to be bound. This is done as shown in Listing 10. As soon as the preprocessing is completed the service can be invoked.

```
// do preprocessing as above
ServiceFrontendFactory fac =
  ServiceFrontendFactory.getFactory
    ("at.ac.tuwien.infosys.dsg.daiosPlugins."+
     "nativeInvoker.NativeServiceInvokerFactory");

ServiceFrontend frontend =
  fac.createFrontend(
    new URL("http://my.service.com/wsdl")
  );

// construct message as above
DaiosMessage message = new DaiosMessage();
message.setString("name", "Philipp Leitner");

// do blocking invocation
DaiosOutputMessage out = frontend.requestResponse(in);
```

Listing 14: Blocking invocation in Daios

A full example is given in Listing 14. Note how simple blocking dynamic Web service invocations are in Daios (as for instance compared to WSIF).

Using client-side asynchrony is nothing more complex than blocking communication. Daios defines the methods `fireAndForget()`, `pollObjectCall()` and `callback()` for non-blocking invocations. Fire and forget invocations are unreliable in nature and cannot return a result. Poll object invocations are non-blocking and immediately return a concrete subclass of `at.ac.tuwien.infosys.dsg.daios.framework.PollObject`. This class defines four important methods: `responseReceived()` and `errorOccured()` can be used to check whether the invocation is already finished or has failed for some reason. The methods `getResult()` and `getError()` fetch the invocation result or a subclass of `java.lang.Exception` in case of an error.

Callback invocations demand for a little more coding on client application side: for this asynchrony style the client application has to provide an implementation of the interface `at.ac.tuwien.infosys.dsg.daios.framework.interfaces.IDaiosCallback`. In this interface the methods `onComplete()` and `onError()` have to be implemented. Daios will call the respective method when the invocation has finished successfully or has failed.

Listing 14 is very minimalistic. More complete and interesting examples of how Daios can be used to invoke existing (as of 18th September 2007) real-world Web services can be found in the Appendix. Appendix I shows an example of a real-world SOAP invocation: the sample invokes a service that takes a German bank identification number as parameter and returns the details of the corresponding bank. This service is implemented using Apache Axis 2 and uses a `document/wrapped` WSDL encoding style. Appendix J on the other hand exemplifies a RESTful invocation. In this example the *Flickr* [35] REST interface [36] is used to retrieve a list of the currently most "interesting" photos. The example invocation needs a Flickr API key which can be ordered on the Flickr website for free. It is often argued that the Flickr REST API is not actually RESTful since it (just like many other well-known REST APIs) does not adhere to some of the rules for RESTful architectures as described in Chapter 2 [65]. For instance the service provides only a single service endpoint address for all invocations and therefore does not adhere to the rule of addressability. Furthermore the API only uses HTTP GET and encodes the "method" information as GET parameter. However, the interface is RESTful enough for the purpose of this demo.

Most of the Daios code is rather self-explanatory, but there are a few pitfalls and limitations to keep in mind:

- In SOAP-based invocations it is possible but usually not necessary to explicitly set the endpoint address of the service to invoke (since the address can be extracted from the WSDL definition anyway). However, RESTful invocations need an explicit endpoint address set. Omitting the endpoint address for REST invocations will result in a runtime error being thrown.
- Daios currently does not set the `SOAPAction` HTTP header correctly in SOAP invocations. If this header is actually checked by the service (usually it is not) then the client application has to set the header explicitly.
- If no interface description is specified when preprocessing is commenced (see for example the REST invocation in Appendix J) then Daios assumes a HTTP GET invocation with URL-encoded parameters. If an example request is specified then HTTP POST with XML-encoded parameters is assumed. There is currently no way of overwriting these conventions for RESTful service invocations. Other HTTP methods such as PUT or DELETE are currently not supported due to their low prevalence in current real-world service implementations.

4.4 Advanced Features

Section 4.3 has explained the basic client-side interface of Daios. This section will detail a few more advanced features that deserve mentioning.

4.4.1 Intercepting the Framework

The Daios framework has an *interceptor* interface [69] that allows client application developers to react to internal events in the framework. That way "services can be added transparently to [the] framework and triggered automatically when certain events occur" [69]. Interceptors may hook well-defined *access points* in order to analyze the internal behavior of Daios, log events or even change the data flow in the framework.

Daios interceptors are subclasses of `at.ac.tuwien.infosys.dsg.daios.framework.DaiosInterceptor`. If only a few events should be hooked it might be advantageous for the client developer to subclass `at.ac.tuwien.infosys.dsg.daios.framework.`

Phase	**Description**
Frontend Creation	Preprocessing (service binding)
Invocation	Service invocation
WSDL Processing	Retrieving and parsing WSDL definitions
WSDL Type Parsing	Parsing XML Schema types
REST example fetching	Loading an example for “REST by example”
Input Conversion	Converting a Daios input message to a stack-specific format
HTTP transfer	The actual service invocation
Output Conversion	Converting the invocation response back from a stack-specific format to Daios message format

Table 7: Interceptable events in Daios

`DefaultInterceptor` instead, the default interceptor which provides an empty default implementation for each hook. Table 7 summarizes all available access points, each one corresponding to important phases in the course of dynamic service invocation as described in the Figures 9, 12 and 14. Each phase issues a hookable event during entry and exit, i.e. there is an entry event and an exit event for each of the phases in Table 7. Some of the phases are inapplicable for some invocation styles, e.g., there is no “WSDL type parsing” phase in a RESTful invocation.

An example implementation of a Daios interceptor which logs all SOAP payload traffic (outgoing requests and incoming responses) to `System.out` is provided as Appendix K.

4.4.2 Fixing Daios Behavior in non-standard Situations

```
// do preprocessing as above
ServiceFrontend frontend = ...

// hard-set endpoint and operation
frontend.setEndpointAddress(new URL("http://my.service.com/epr"));
frontend.setWSDLOperationName(new QName("helloWorld"));

// ... continue invocation as usual
```

Listing 15: Hard-coding WSDL parameters in Daios

For most cases the general frontend interface as used in the previous examples is well suited, but there are situations where the application developer wants to have more control over Daios' behavior. For instance the developer may want to choose an endpoint address that differs from the one defined in the WSDL definition of the service, or he may want to override the distance calculation algorithm of the mapper and explicitly specify the WSDL operation to use. Listing 15 shows an example of this functionality. The resulting application will be (very slightly) faster, but also rather tightly coupled to the service provider. Using this functionality should therefore be avoided if possible.

For more complex modifications of Daios' behavior it is possible to directly access the underlying invocation backend. One example is displayed in Listing 16: the Axis 2 backend is accessed and configured to not chunk the request body.

```
// do preprocessing as above
ServiceFrontend frontend = ...

// set Axis 2 options to use
Options opts = new Options();
opts.setProperty(
  org.apache.axis2.transport.http.HTTPConstants.CHUNKED,
  Boolean.FALSE);
((Axis2ServiceInvoker)frontend.getImplementor())
  .setAxis2Options(opts);

// ... continue invocation as usual
```

Listing 16: Accessing the invocation backend in Daios

Accessing the backend as in Listing 16 is powerful. It allows application developers to use the sophisticated possibilities of the Axis 2 SOAP stack and tweak the invocation in various ways. However, the disadvantage is that using specific backend functionality ties the implementation to the backend - switching to e.g., the native backend is not so easy anymore, and much of the flexibility provided by the CPWSI structure of Daios is lost.

It is also possible to access the SOAP and HTTP *header fields* if this is necessary. The native backend provides full support in this respect: it allows to set SOAP and HTTP headers for the invocation request, and provides access to all SOAP and HTTP response headers. The Axis 2 backend is a little more restricted: here only the SOAP request headers may be set. Listing 17 gives an example of how to work with SOAP and HTTP header fields using the native backend: a new SOAP header "mustUnderstand" and a HTTP header "SOAPAction" are introduced, and the value of the HTTP response header "Server" (the server identification string) is printed to `System.out`.

```
// do preprocessing as above
ServiceFrontend frontend = ...

// add mustUnderstand SOAP header
((NativeServiceInvoker)frontend.getImplementor())
  .addHeader(new QName("mustUnderstand"), "false");

// add new SOAPAction HTTP header
((NativeServiceInvoker)frontend.getImplementor())
  .addHTTPHeader("SOAPAction", "urn:helloWorld");

// ... continue invocation as usual

// Display the server ident string of the service provider
System.out.println(
  ((NativeServiceInvoker)frontend.getImplementor())
    .getResponseHTTPHeaders().get("Server")
);
```

Listing 17: SOAP and HTTP headers in Daios

A few HTTP headers should not be changed: client applications for instance should not override the `Transfer-Encoding` HTTP header; Daios will transfer HTTP requests chunked anyway, and if the header is set inaccurately the server may fail to process the invocation correctly.

5 Evaluation

Better than before.
Better, stronger, faster.
– From the movie "The Six Million Dollar Man"

Chapter 4 has described the implementation of the Daios framework for dynamic Web service invocation. This chapter will now analyze the *performance* of Daios in depth: it will give insight into the internal processing of Daios and compare its performance to various well-established frameworks. Three different aspects of performance will be evaluated:

- *Functionality*: What are the defining features of the framework in question? What Web service standards does the framework support?
- *Response Time*: How long is the time span between issuing a request and receiving the result using the framework in question? How long does every individual step in an invocation take?
- *Memory Consumption*: How much memory does a framework use at most during an invocation? How much memory is used in any step of an invocation?

In order to get a clearer view at the results of the performance comparison this Chapter will start off by analyzing the internal processing of Daios in detail: it will be explained what the most important phases of dynamic invocations (cp. Chapter 4) with regard to runtime performance are, and depict how memory consumption is distributed over the course of an invocation. Afterwards Daios will be compared to four other Web service frameworks: Apache WSIF (see Section 3.1.1), Apache Axis 2 (Section 3.1.2), Codehaus XFire and Apache CXF (both introduced very briefly in Section 3.1.4).

Please note that this thesis will only compare dynamic Web service invocations, and will not research general Web service performance (for instance as compared to distributed object middleware). This topic has already been extensively covered in the community (see for instance [19, 22, 27]). Just as little will this thesis be covering the various possibilities of improving Web service performance (for instance by using MTOM [88], XML compression [16] or differential SOAP serialization [1] and deserialization [75]). Addressing any of these topics would deserve a thesis on its own and has therefore been omitted.

5.1 Evaluation Scenario

All performance tests have been carried out in the same environment:

- Test machine was an AMD Athlon64 4000+ desktop PC with 1024 MB RAM. The machine was running on top of Ubuntu Linux 7.04, with a Linux kernel version 2.6.20 for the AMD64 architecture.
- The test machine was using the standard Sun JRE (Java Runtime Environment), version 1.6.0 for Linux, and all code was compiled using the Sun Java compiler `javac`, version 1.6.0 for Linux.
- The test Web services have been deployed on an Apache Tomcat application server, version 5.5.17, using either Apache Axis 1.4 (for `RPC/encoded` services) or Apache Axis 2, version 1.2 (for `document/wrapped` services). The Tomcat server was running on the same machine as the test clients in order to keep the impact of network latency low.
- Two test services with three operations have been implemented and deployed. The service implementations have been kept minimal: every operation takes one single argument (a String, an array of Strings or base64-encoded binary data) and simply returns the given argument to the client. These three operations have been deployed on Apache Axis using the `RPC/encoded` WSDL encoding style and on Apache Axis 2 using `document/wrapped`. The operation with a String argument will be referred to as the "String operation", the operation with the array argument as the "array operation" and the operation with the binary argument as the "binary operation" during the remainder of this chapter.

During the test runs the host machine was not used for anything else, i.e. all applications not necessary for the tests were closed prior to starting the tests. Runtime tests are subject to unpredictable fluctuations. All runtime tests have therefore been repeated 250 times and averaged. Time has been measured using the Java `System.currentTimeInMillis()` method, and memory usage has been monitored using the `javax.management` API. The memory consumption of the Java Virtual Machine (JVM) itself has been subtracted from the memory to gain a more accurate measure of the memory overhead of the frameworks. Unless stated otherwise all Daios tests have been carried out using SOAP and the native backend.

5.2 Detailed Analysis

The following Section will examine the Daios performance on *phase level*. The phases used during the evaluation are identical to the ones presented in Chapter 4, but sometimes phases will be omitted or combined in order to produce a clearer picture.

5.2.1 Internal Processing

First of all a look at the runtime behavior of Daios will be presented: this Section will clarify what the most expensive (in terms of runtime) phases during different types of dynamic invocations are. All tests in this Section refer to invocations of the "String operation", but the findings below are also applicable to other types of operations.

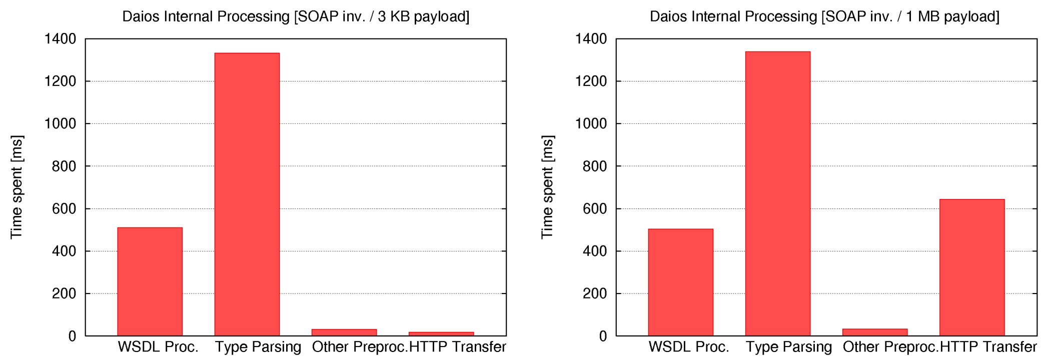


Figure 17: Daios runtime performance - SOAP invocation

Figure 17 displays the most relevant phases during a SOAP-based dynamic Web service invocation. In the left part of the figure a single SOAP invocation with a rather small payload is conducted. In such a situation the preprocessing is extremely dominant: about 99% of the overall invocation time is spent in the preprocessing phase. Within this phase the XML type parsing is the more expensive part (about 70%). Other phases such as input and output conversion are not significant and are therefore omitted in the figure. The right part shows a single dynamic invocation with a more sizeable payload (about 1 MB). As expected the preprocessing time is constant for any payload size, but the time to carry out the actual invocation (the time spent in the invocation phase) increases. This is mostly due to an increased HTTP transfer time. For this bigger payload the preprocessing share of the total invocation time is lower, but still very significant (roughly 75%).

However, preprocessing is only necessary for the first invocation of a certain service.

All subsequent invocations can reuse the first service binding. That is, the preprocessing share of the total invocation time decreases with the number of invocations targeting the same service. It is therefore advisable for client application developers to target as many invocations to the same service as possible to minimize the overhead resulting from WSDL processing.

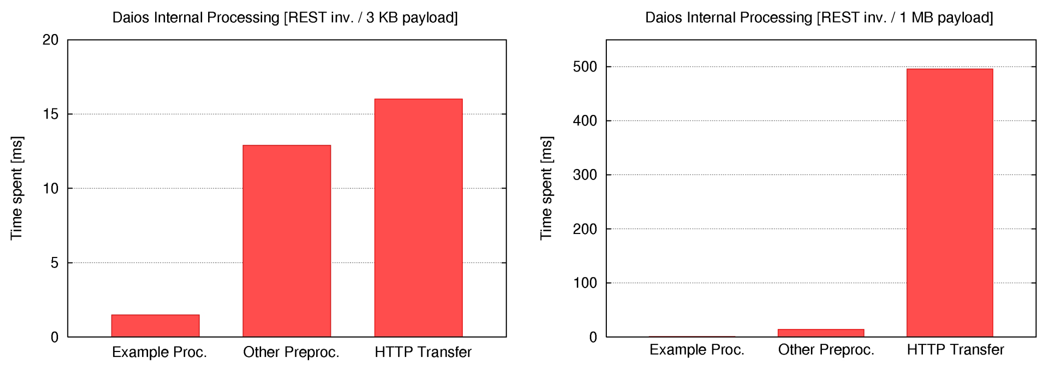


Figure 18: Daios runtime performance - REST invocation

Figure 18 shows that the situation is entirely different for RESTful service invocations. In REST preprocessing is a negligible factor: there is no formal interface definition comparable to WSDL to compile in this step, and loading an example for "REST by example" does not cause a big overhead. Even for single small invocations preprocessing is not a dominant factor in RESTful invocations. The phase denoted "Other Preproc." in the figure describes a collection of all other preprocessing activities besides example loading, for instance setting up the service invoker.

The overall invocation time for a single RESTful invocation with small payload is remarkably short: about 35 ms as compared to 1900 ms in SOAP. For bigger payloads the preprocessing effort becomes more and more dispensable. The actual invocation can also be handled more efficiently in REST-based invocations: for big payloads the SOAP invocation phase takes about 650 ms as compared to only 490 ms in REST. The reason for this difference is the additional overhead entailed by the SOAP protocol as compared to the quite naive REST way of data transmission.

However, the shorter preprocessing phase in REST invocations does not come without a cost. Since REST is lacking a strong formal interface definition language the client application developer has to know a lot of details about the service beforehand, and has to code this knowledge into the service client. This means that current REST Web service clients are much more coupled to their service provider as SOAP-based ones. It also means that the reduced runtime during the preprocessing is probably not actually saved, but only shifted to the client application.

5.2.2 Memory Consumption

A different performance measure is memory consumption. In this Section the memory footprint of Daios will be analyzed on phase level. The total memory consumption will be separated into two parts, the *heap memory* and the *stack memory*. In Java heap memory denotes the memory area where instantiated objects reside, while native types such as `ints` or `shorts` are stored on the stack. The following tests refer to using the "String operation" with a rather big payload of about 1.49 MB.

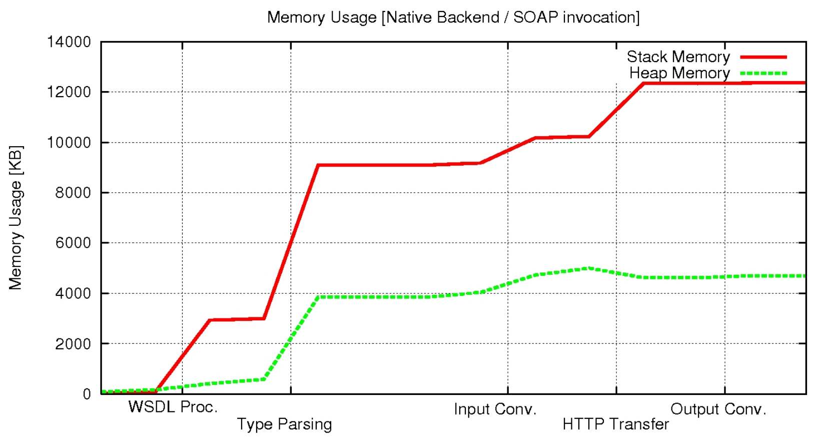


Figure 19: Daios memory consumption - SOAP invocation

Figure 19 shows the memory consumption during a SOAP-based invocation. During such invocations the memory consumption increases dramatically when the WSDL definition and XML type system are parsed, and again when the Daios input message is converted and the HTTP response is received. The size of the later peaks is mostly dependent on the payload size of the invocation. The overall memory consumption is relatively high since a lot of service information has to be kept in memory.

REST invocations (figure 20) do not need so much memory since they do not require a compiled interface definition. During RESTful invocations the memory usage increases when user input is converted and when the HTTP response is received.

Note that the size of the payload is the only really significant factor for REST-based invocations: before issuing the invocation the overall memory consumption is about 1500 KB (roughly equal to the payload size), and after the response is received memory consumption rises to little more than 3000 KB, i.e. two times the payload

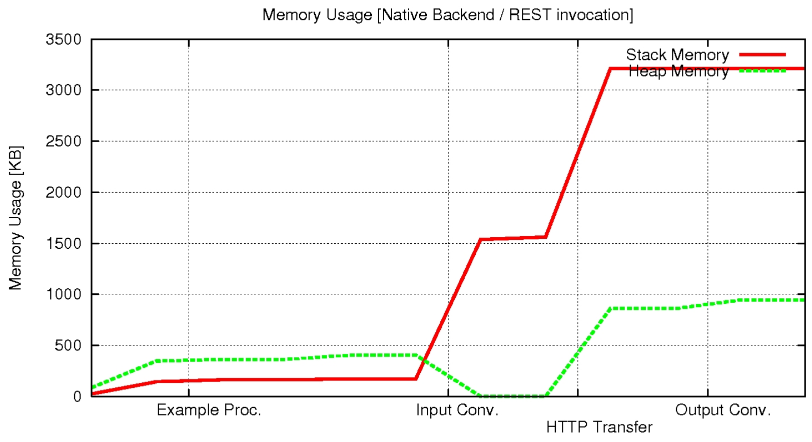


Figure 20: Daios memory consumption - REST invocation

size. The memory overhead of the Daios framework for REST invocations seems to be negligible.

5.3 Comparison to other Frameworks

Section 5.2 has given a relatively fine-grained view on Daios' performance. Now the scope will be shifted to a more holistic view of dynamic invocations, and compare the framework to other well-known Web services stacks: Apache WSIF, Apache Axis 2, Codehaus XFire and Apache CXF. In this Section the frameworks will be examined as "black boxes", i.e. the internal structure of the candidates will not be considered.

5.3.1 Functional Analysis

This Section will compare the frameworks with respect to the first performance criterion, *functionality*. It will detail the features and standards that the frameworks support. To keep the comparison simple it is necessary to confine to evaluating only the *existence* of certain features, and not so much their *quality*. Exceedingly good or bad particular cases are commented in the textual descriptions.

Table 8 lists the frameworks' support for standard Web services features and protocols. A "✓" means that the framework includes support for the respective feature. The Table shows that WSDL 2.0 has not yet caught on: only Axis 2 supports the new WSDL standard so far. CXF has announced WSDL 2.0 support, but it is not included in the first release of the framework. `RPC/encoded` has become very unpopular since “forbidden” by the WS-I basic profile: only the older test candidate WSIF and the Daios framework support this encoding style. Supporting at least the most important WS-* protocols, i.e. WS-Security, WS-Addressing and WS-Policy, as well as MTOM for efficient SOAP transmission, is standard for industry-strength Web services stacks. It is also observable that all younger frameworks have embraced XML stream parsing for efficient handling of XML data.

Feature	**Daios**	**WSIF**	**Axis 2**	**XFire**	**CXF**
SOAP:					
1.1	✓	✓	✓	✓	✓
1.2	✗	✗	✓	✓	✓
WSDL:					
1.1	✓	✓	✓	✓	✓
2.0	✗	✗	✓	✗	✗
Encoding:					
RPC/encoded	✓	✓	✗	✗	✗
doc/literal	✗	✓	✓	✓	✓
doc/wrapped	✓	✓	✓	✓	✓
Transport:					
HTTP	✓	✓	✓	✓	✓
JMS	✗	✓	✓	✓	✓
SMTP	✗	✗	✓	✗	✗
local	✗	✓	✗	✗	✓
TCP	✗	✗	✓	✗	✗
EJB	✗	✓	✗	✗	✗
WS-*:					
WS-Addressing	✗	✗	✓	✓	✓
WS-Security	✗	✗	✓	✓	✓
WS-Policy	✗	✗	✓	✗	✓
Other:					
StAX	✓	✗	✓	✓	✓
MTOM	✗	✗	✓	✓	✓

Table 8: Web service standards support

It is obvious that the Daios framework falls behind the other candidates in this comparison, particularly compared to Apache Axis 2 and CXF. Daios is (as opposed to all other candidates examined here) no industry-strength product but a research prototype, and up to now more emphasis has been put on dynamic invocation than on already well-understood state of the art features. However, it is important to note that none of these missing features poses a conceptual problem, and further versions of Daios may well include support for SOAP 1.2, WS-Security or WS-Policy.

Feature	**Daios**	**WSIF**	**Axis 2**	**XFire**	**CXF**
DII:					
simple types	✓	✓	✓	✓	✓
arrays of simple types	✓	✓	✓	✓	✓
complex types	✓	✗	✓	✗	✓
arrays of complex types	✓	✗	✓	✗	✓
Client-side asynchrony:					
synchronous	✓	✓	✓	✓	✓
fire and forget	✓	✗	✓	✗	✓
callback	✓	✗	✓	✗	✓
poll object	✓	✗	✗	✗	✓
Client-side REST:					
HTTP GET	✓	✗	✓	✗	✓
HTTP POST	✓	✗	✓	✗	✓
HTTP DELETE	✗	✗	✓	✗	✓
HTTP PUT	✗	✗	✓	✗	✓
Rest by example	✓	✗	✗	✗	✗
WADL	✗	✗	✗	✗	✗

Table 9: Dynamic invocation features

Table 9 focuses more on non-standard features that are relevant with regard to dynamic service invocation.

The first block "DII" considers the expressiveness of the dynamic invocation interface of the candidates. Daios and CXF support simple and complex types as well as arrays of both. Complex type support in the XFire DII has been a long-discussed issue, but to the best knowledge of the author this issue has never been resolved until the XFire project was merged with Celtix. Apache Axis 2 has a fully expressive DII, but it demands for the client application to construct the XML payload of the SOAP message itself (cp. Section 3.1.2), hence offering very limited support for dynamic invocations.

Client-side asynchrony is relatively wide-spread in the Web services community today. All newer frameworks provide at least some non-blocking invocation facilities.

REST support is also a property of the newer frameworks: WSIF as well as XFire do not contain any support for RESTful services while Apache Axis 2 and Apache CXF exhibit a rather complete set of REST features. "REST by example" is a distinguishing feature of Daios and not supported by any other candidate so far. The REST interface description language WADL is not supported by any candidate today.

The question whether a framework supports REST or not is often discussed on an ideological level: it is often argued that frameworks such as Apache Axis 2 do not really support REST, since they often reduce REST to “XML over HTTP”, ignoring the “REST ideology” entirely. For this thesis this ideological debate has been avoided on purpose, instead the focus has been set on purely technological issues.

5.3.2 Comparison of Runtime Performance

This Section will measure the runtime behavior of the test candidates. It will detail what response times client applications can expect when issuing dynamic invocations using the candidate frameworks.

As explained in detail in Chapter 4 dynamic invocations split up in two parts: the preprocessing, when the key service data is collected and compiled, and the actual invocation, when a SOAP request is disposed and the response received. Comparing the DIIs of the candidates is difficult since they are variably developed and require a different amount of work in the client application. This evaluation therefore concentrates on comparing Daios to WSIF, Axis 2, XFire and CXF used for *ad hoc* dynamic invocations similarly to the procedure described in Section 3.1.8 or [51]: static stubs are generated at run-time in the preprocessing phase of the invocation; these stubs are then used to carry out the actual invocation.

In the following comparison the preprocessing and the invocation are separated: firstly the performance of the preprocessing step will be compared, afterwards the actual invocation times will be measured.

Preprocessing. All preprocessing in Daios takes place when a new frontend is created using the `createFrontend()` method. To measure the Daios preprocessing time it is therefore sufficient to meter the time that Daios spends in this method during an invocation. For all other frameworks it is sufficient to launch the standard WSDL-to-code compiler of the candidate, and measure how long it takes to create the static stubs. For this comparison the overhead that would be introduced by loading the static stubs into the classpath can be ignored since this overhead is extremely small in comparison to the time necessary for stub generation. In this step the WSIF framework cannot be evaluated since it does not provide a WSDL compiler of its own.

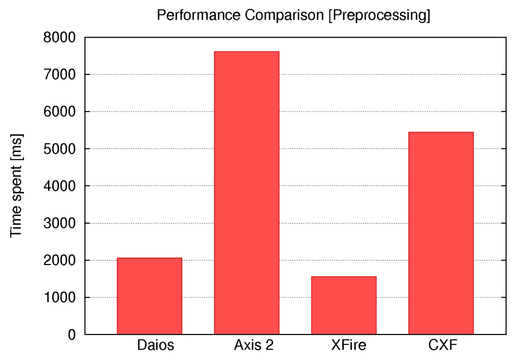


Figure 21: Comparison of preprocessing costs

Figure 21 compares the preprocessing / code generation times of the test candidates. Preprocessing in Daios and XFire is relatively fast, about 4 times as fast as code generation in Apache Axis 2. Interestingly the CXF framework takes almost three times longer to generate static stubs as its predecessor XFire. Even though Daios is comparatively fast in this respect it should not be discarded that about 2 seconds on a standard desktop computer still is a very large overhead for dynamic invocations.

Invocation Time. In the long run the actual invocation time is more important than the preprocessing overhead (which optimally occurs only once per service). This Section will compare the invocation performance of the test candidates.

All three operations as described in Section 5.1 are used to evaluate the frameworks; every test will be conducted once using `RPC/encoded` and once using `document/wrapped` encoding. For `RPC/encoded` only Daios and Apache WSIF are evaluated since the other frameworks do not support this encoding style. For every test case the time necessary to conduct the actual invocation has been measured against linearly increasing payload size.

In theory the dependency between payload size and invocation time should be linear, but in practice implementation details (e.g., internal buffer sizes and similar) lead to figures which are hard to read and interpret. For the following Section the data has therefore been linearized through *linear regression*. The original (not linearized) plots are given as appendix L.

Figure 22 details the linearized performance data for simple string invocations. The left part of the figure shows the results of the `RPC/encoded` tests. Daios is faster than

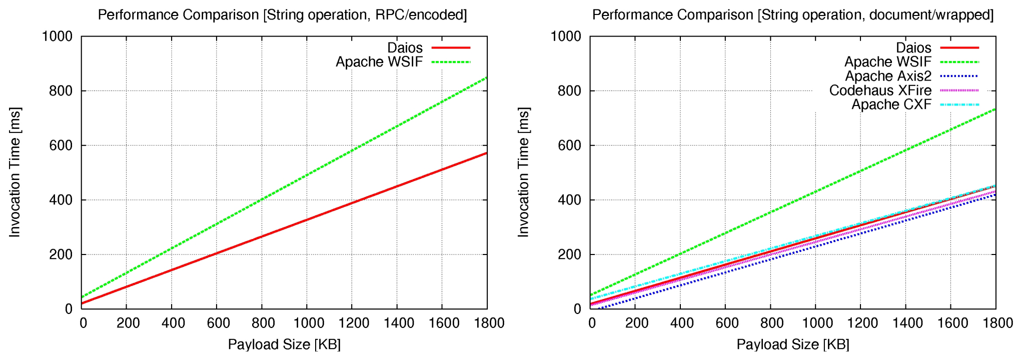


Figure 22: Comparison of simple invocations - linear regression

WSIF for any invocation size, and the difference is getting bigger with increasing payload. For 1800 KB invocations the performance difference is already about 250 ms per invocation. Looking at `document/wrapped` invocations (right side) one can discover that Apache Axis 2 has the best runtime performance in this test, but the differences to XFire, Daios and CXF are only marginal and practically constant with growing payload size. WSIF is far behind in that test. Both Daios and WSIF are faster in `document/wrapped` mode. This is to be expected and reflects the additional overhead introduced by `RPC/encoded` encoding.

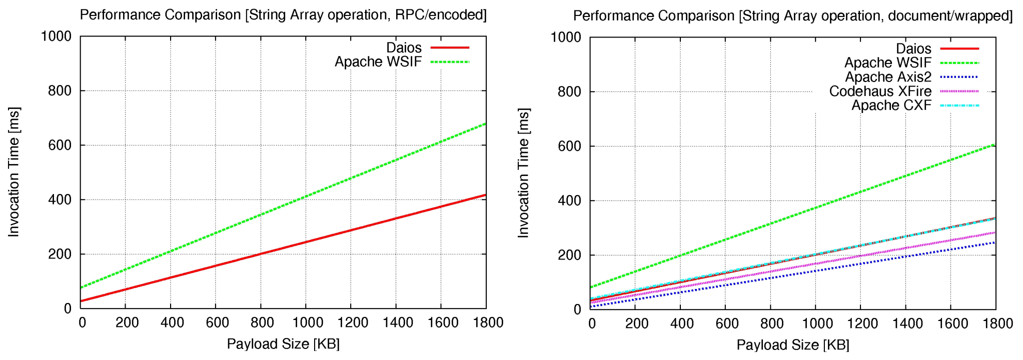


Figure 23: Comparison of array invocations - linear regression

Figure 23 shows the performance data for array invocations. For `RPC/ encoded` (left graph) WSIF is again a lot less performant than Daios. The right graph (`document/wrapped`) shows a clearer picture than figure 22: Apache Axis 2 is again the most performant candidate, slightly faster than Codehaus XFire. The mid-field is formed by Daios and Apache CXF, already with considerable offset (about 100 ms between Axis 2 and Daios or CXF). Apache WSIF is again by far the slowest framework in the test field.

Figure 24 depicts the same for the last test operation, the "binary invocation". For

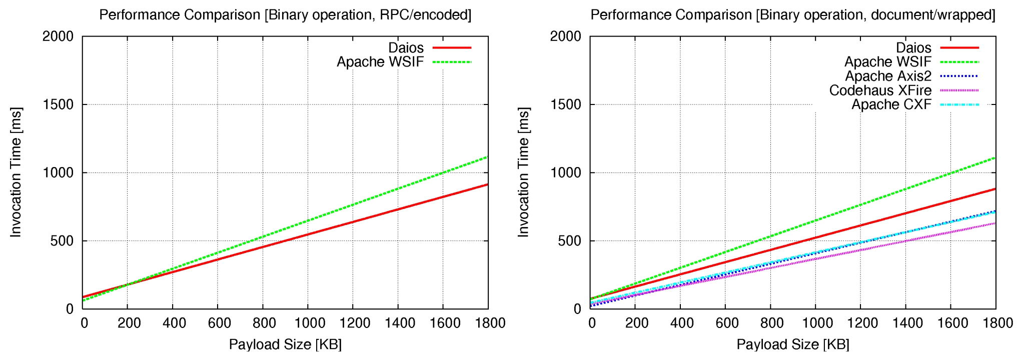


Figure 24: Comparison of binary invocations - linear regression

`RPC/ encoded` invocations (left part) Daios is again faster than WSIF, but the difference is less dramatic as compared to the other test operations. For `document/wrapped` invocations (right part) Codehaus XFire is decidedly the best candidate, followed by Apache Axis 2 and Apache CXF. Daios is a little behind the best frameworks in this test, but still a good deal before Apache WSIF. Note that the overall invocation times are a lot higher for binary invocations because of the additional processing time necessary for base64 encoding.

It can be concluded that Apache Axis 2 is the fastest test framework in the candidate field, but the differences to Codehaus XFire, Apache CXF as well as Daios are only marginal. Only Apache WSIF falls behind, most probably because of the age of the framework. Another interesting fact is that XFire seems to be a little faster than its successor CXF at the moment, but CXF is still in a rather early development stage and its performance may improve during the next months and years.

5.3.3 Memory Consumption

This Section will compare the candidate field with regard to the last performance indicator, the memory footprint. Most important in this respect is the highest total memory consumption, that is the maximal value of heap and stack memory in use at any time during the invocation.

Figure 25 shows the measured maxima. Daios has the highest memory consumption of all frameworks in the test, followed by Apache Axis 2 and Codehaus XFire. Apache WSIF as well as Apache CXF exhibit a relatively small memory footprint. However, with a memory footprint no smaller than 10000 KB none of the test candidates is particularly memory-efficient, and presumably none of candidates is usable for areas such as mobile and ubiquitous computing [93] where memory is a sparse ressource.

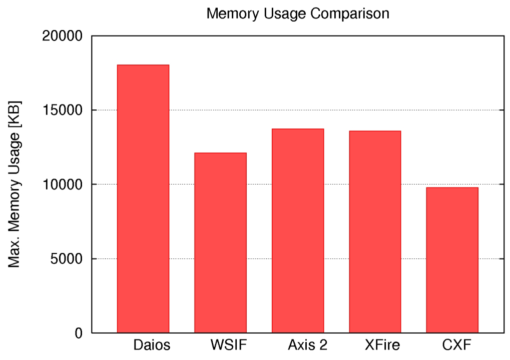


Figure 25: Comparison of maximum memory consumption

Daios' bad memory footprint in SOAP invocations is a result of the WSDL and XML Schema parsing necessary during the preprocessing phase. For REST invocations (which do without WSDL and XML Schema parsing) Daios has a rather small memory footprint (cp. Section 5.2).

5.4 Evaluation Summary

Chapter 1 of this thesis has defined 6 requirements for a client-side Web service framework that truly supports the SOA vision:

- The goals "stubless service invocation", "message-driven", "protocol-independent" and "support for asynchronous communication" have surely been accomplished. Daios can only be used through a DII using a messsage-based communication paradigm, supports both SOAP (`document/wrapped` as well as `RPC/encoded`) and REST, and allows for various types of non-blocking invocations.
- Arguably the goal "simple API" has also been accomplished. The Daios client-side API as described extensively in Chapter 4 is simple to use and does not require any deeper knowledge of XML or even Web service technology. Nonwithstanding, advanced users can still control the behavior of the framework in more detail if they need to.
- Section 5.3 showed that the goal "acceptable runtime behavior" has also been reached. The runtime performance of the Daios framework is significantly better than WSIF's, and for all practical purposes on one level with the best frameworks available today. Using Daios does therefore not result in a significant performance penalty.

5.4.1 Limitations

For all the advantages described above the current version of Daios also has a few limitations:

- Daios is a research prototype. The implementation has therefore been centered on the design goals described above, and only a subset of current Web service standards and state of the art features have been implemented. Other existing frameworks are of course supporting a much broader set of Web service standards and features than Daios.
- Daios is strictly intended for dynamic service invocations. For static scenarios lacking any dynamics other frameworks are much more appropriate.
- Even though Daios supports both SOAP and REST the feature set for RESTful services is not yet as complete as for SOAP. Daios currently supports only HTTP GET and POST, and would need to process a REST interface definition language in order to bring REST and SOAP support to the same level. Unfortunately no such REST interface definition language is wide-spread enough yet.

6 Conclusion and Future Work

Come, my friends,
'Tis not too late to seek a newer world.
To sail beyond the sunset, and the baths
Of all the western stars, until I die.
– Alfred Lord Tennyson, "Ulysses" [78]

The Service-Oriented Architecture vision expects distributed systems to use the "triangle of *publish-find-bind*" to create a loosely coupled architecture, where all software components provide services to the system as a whole. All services are either atomic or composed of other services, and are independent from each other. Services may be selected or substituted at run-time. Unfortunately such a system is hard to implement today: service registries are not yet sophisticated enough to allow for run-time service selection based on service semantics or quality-of-service, and state of the art client-side service frameworks are not well-suited for run-time service binding. These service frameworks (including for instance Apache Axis 2, Codehaus XFire or Apache CXF) are usually used through static stubs, and are therefore tightly coupled to service providers. Current dynamic invocation interfaces are more like the "poor cousins" of the stub interfaces, and struggle with fundamental problems which make them hard to use for SOA scenarios. Support for non-blocking and document-based communication is increasing in the service community, but RPC-style programming models continue to prevail.

In order to bypass these issues practitioners are often forced to employ workarounds: sometimes static stubs are generated at run-time to implement systems that are able to arbitrarily change service providers. However, creating static stubs at run-time is cumbersome and expensive in terms of performance.

The Daios (Dynamic and asynchronous invocation of services) framework aims at providing a client-side service framework that is better suited for such a SOA: Daios provides a fully expressive and easy to use DII, works entirely message-oriented and has full support for non-blocking communication. The Daios framework has been implemented in the Java programming language, taking into account state-of-the-art development techniques such as design and architectural patterns, and using well-established open-source libraries as for instance Apache AXIOM, Codehaus Woodstox, Apache XMLBeans or Apache HTTPClient. Daios is implemented in a very extensible way: new backends providing support for new standards and service protocols can be introduced easily, possibly even at run-time of a client application.

Daios is therefore very well suited for implementing long-running, loosely-coupled distributed systems that want to fully utilize the power of SOA.

The evaluation presented in chapter 5 of this thesis has proven that Daios (although being a research prototype) is on one level with widely used Web service frameworks considering runtime performance. Daios is considerably faster than Apache WSIF (which is considered the best dynamic service invocation framework so far), and only marginally slower than Apache Axis 2 or Apache CXF. If Axis 2 or CXF is used "pseudo-dynamically" as mentioned above, then the Daios solution is clearly superior: the Daios programming model is much more intuitive than using dynamically generated stubs through reflection, and Daios preprocessing is faster than generating static stubs at run-time.

Summarizing it can be stated that the Daios concept seems promising enough to solve the issues of dynamic service invocation. Preliminary evaluation of the framework has proven that using the Daios framework is not inefficient for practical purposes, and that the simple message-based interface allows for a much more natural programming model in the client application.

6.1 Future Work

However, there are a few open issues in the Daios framework left for future work: perhaps most important is the inclusion of more state of the art features into the Daios framework in order to make it more apt for practical use. On top of the currently missing feature list would be support for wide-spread and important WS-* standards, in particular for WS-Security and WS-Addressing, as well as support for the latest WSDL and SOAP standards. Another missing feature that would be important for the Daios concept is a strong support for WS-Policy - Daios should be able to abstract transparently from different WS-Policy policies and with as few input from the client application as possible.

Preprocessing is currently the weakest point of the framework in terms of performance. Further work could probably be done to reduce the time spent in the preprocessing phase (e.g., by tuning the WSDL and XML Schema processing components). Another possible optimization of Daios would be to persist service bindings to a mass storage device - in that case the expensive preprocessing needs to be done only once per service. If service bindings are to be persisted a change detection mechanism for service interface definitions has to be in place so that Daios is able to detect automatically whether a bound service has changed and needs re-binding.

Furthermore, the general REST support of Daios deserves further working on: probably most importantly Daios should at some point support all HTTP operations (i.e. also including HEAD and OPTIONS) and WADL or whatever interface definition language catches on in the REST community.

Finally implementing a few more default backends would be highly deservable, bringing the addional power of e.g., XFire of CXF to Daios. It would also be interesting to further investigate whether it is possible to change the backend entirely without re-binding the service - such a feature would increase the overall flexibility of the framework dramatically since it would be possible to select a suiting backend individually for every single invocation.

Appendix

A List of Abbreviations

API	Application Programming Interface
AXIOM	AXis Object Model
CORBA	Common Object Request Broker Architecture
CPWSI	Composite Pattern for stubless Web service Invocation
Daios	Dynamic and asynchronous invocation of services
DCOM	Distributed Component Object Model
DII	Dynamic Invocation Interface
EAI	Enterprise Application Integration
ebXML	electronic business over XML
EJB	Enterprise Java Beans
ESB	Enterprise Service Bus
HTTP	Hypertext Transfer Protocol
IDL	Interface Definition Language
IT	Information Technology
JAX-RPC	Java API for XML-based RPC
JAX-WS	Java API for XML-based Web Services
JEE	Java Enterprise Edition
JMS	Java Messaging Service
JRE	Java Runtime Environment
JSON	JavaScript Object Notation
JSR	Java Specification Request
JVM	Java Virtual Machine
LAN	Local Area Network
MEP	Message Exchange Pattern
MOM	Message Oriented Middleware
OASIS	Organization for the Advancement of Structured Information Standards
OMG	Object Management Group
OOP	Object-Oriented Programming
PC	Personal Computer
POSA	Pattern-oriented Software Architecture
REST	Representational State Transfer
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SAIWS	Simple Asynchronous Invocation Framework for Web Services
SC	Sequencing Constraints
SOA	Service-Oriented Architecture

Continued on Next Page…

SoC	Service-oriented Computing
SMTP	Simple Message Transfer Protocol
SSDL	SOAP Service Description Language
StAX	Streaming API for XML
TCP	Transport Control Protocol
UDDI	Universal Description, Discovery and Integration
UML	Unified Modelling Language
URI	Universal Ressource Identifier
URL	Universal Ressource Locator
WADL	Web Application Description Language
WAN	Wide Area Network
WS-BPEL	Web Service Business Process Execution Language
WS-CDL	Web Service Choreography Description Language
WSDL	Web Service Definition Language
WS-I	Web Services Interoperability Organization
WSIF	Web Services Invocation Framework
WSMQ	Web Services Message Queue
WWW	World Wide Web
XML	eXtensibel Markup Language

Table 10: List of Abbreviations

B SOAP RPC Example

```
<?xml version='1.0' encoding='UTF-8'?>
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
>
  <soapenv:Header />
  <soapenv:Body>
    <getAddress>
      <in2 xsi:type="xsd:string">1150</in2>
      <in0 xsi:type="xsd:string">Philipp Leitner</in0>
      <in4 xsi:type="xsd:int">14</in4>
      <in1 xsi:type="xsd:string">Wien</in1>
      <in5 xsi:type="xsd:int">15</in5>
      <in3 xsi:type="xsd:string">Sperrgasse</in3>
    </getAddress>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 18: SOAP-encoded RPC call

C Complete WSDL Example

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
  xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
  xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
  xmlns:ns0="http://my.namespace.com/types"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  targetNamespace="http://infosys.tuwien.ac.at/dacoss/eval/doc/">

  <!-- type definitions -->
  <wsdl:types>
    <xs:schema xmlns:ns="http://my.namespace.com/types"
       attributeFormDefault="qualified" elementFormDefault="qualified"
       targetNamespace="http://my.namespace.com/types">
      <xs:element name="helloWorld">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="firstParam" nillable="true"
               type="xs:string" />
            <xs:element name="secondParam" nillable="true"
               type="xs:string" />
            <xs:element name="thirdParam" nillable="true"
               type="xs:string" />
            <xs:element name="fourthParam" nillable="true"
               type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="helloWorldResponse">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="return" nillable="true"
               type="xs:string" />
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:schema>
  </wsdl:types>

  <!-- message definitions -->
  <wsdl:message name="helloWorldMessage">
    <wsdl:part name="parameters" element="ns0:helloWorld" />
  </wsdl:message>
  <wsdl:message name="helloWorldResponse">
    <wsdl:part name="parameters" element="ns0:helloWorldResponse" />
  </wsdl:message>



  <!-- port type definitions -->
  <wsdl:portType name="HelloWorldServicePortType">
    <wsdl:operation name="helloWorld">
      <wsdl:input
         xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl"
         message="ns0:helloWorldMessage" wsaw:Action="urn:helloWorld"
      />
      <wsdl:output message="ns0:helloWorldResponse" />
      </wsdl:operation>
   </wsdl:portType>

   <!-- binding definitions -->
   <wsdl:binding name="HelloWorldBinding"
      type="ns0:HelloWorldServicePortType">
     <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
        style="document" />
        <wsdl:operation name="helloWorld">
          <soap:operation soapAction="urn:helloWorld"
             style="document" />
          <wsdl:input>
            <soap:body use="literal" />
          </wsdl:input>
          <wsdl:output>
            <soap:body use="literal" />
          </wsdl:output>
        </wsdl:operation>
   </wsdl:binding>

   <!-- service definitions -->
   <wsdl:service name="HelloWorldService">
     <wsdl:port name="HelloWorldBinding"
        binding="ns0:HelloWorldBinding">
        <http:address location="http://my.test.service.com/epr" />
     </wsdl:port>
   </wsdl:service>
 </wsdl:definitions>
```

Listing 19: Complete WSDL Example

D Complete SSDL Example

```
<contract xmlns="urn:ssdl:v1">

  <!-- define types using XML Schema -->
  <schemas>
    <schema>
      <element name="myType" type="anyType" />
    </schema>
  </schemas>

  <!-- define messages -->
  <messages>
    <message name="InMessage">
      <header ref="myType" />
    </message>
    <message name="OutMessage">
      <header ref="myType" />
    </message>
  </messages>

  <!-- define protocols -->
  <protocols targetNamespace="urn:service:sc">
    <protocol name="exampleProtocol">
      <sc>
        <sequence>
          <msgref ref="InMessage" direction="in" />
          <msgref ref="OutMessage" direction="out" />
        </sequence>
      </sc>
    </protocol>
  <protocols>

  <!-- define endpoints -->
  <endpoints xmlns:wsa="http://www.w3.org/2004/12/addressing">
    <endpoint>
       <wsa:Address>http://my.example.com/epr</wsa:Address>
    </endpoint>
  </endpoints>

</contract>
```

Listing 20: Complete SSDL example

E Structural Distance Calculation

```
// proc.: Distance Calculation Algorithm
// in: WSDLMessage or WSDLPart (handled equivalently), DaiosMessage
// out: distance of WSDLMessage to DaiosMessage
//      (NOT DaiosMessage to WSDLMessage!)

distance = 0

if(WSDLMessage no input message)
  return Integer.MAX

else
  forall fields as field in DaiosMessage:
    if(WSDLMessage does not contain part with corresponds(part,field) )
      // the DaiosMessage contains more information
      // than the WSDL message
      // --> incompatible
      return Integer.MAX
    else
      if(part is simple)
        // the part and the fields are totally equivalent
        // --> no distance
        distance += 0;
      else
        // the part is complex
        // --> probably compatible, but we don't know yet,
        //     recursively go deeper
        distance += Distance Calculation Algorithm(part,field)
      mark part used (and never use again)

distance += # of not marked parts in WSDLMessage
return distance

-----------------------------------------------------------------------------

// proc.: corresponds
// in: Part of WSDLMessage part, Field of DaiosMessage field
// out: true or false

if(field.name equal part.name
     AND
   field.type equal part.type)

  return true

else
  return false
```

Listing 21: Structural distance calculation in pseudo-code

F REST by Example Procedure

```
// proc.: REST by example algorithm
// in: example in XML notion, DaiosMessage
// out: XML-encoded REST message

// go over each element and attribute in the example
forall elements as element in the example

  // check whether the element / attribute is contained
  // in the Daios message;
  // if it is and simple add the element and take the
  // value for the element from the DaiosMessage field;
  // if it is not simple construct the complex message
  // part recursively
  if(element contained in DaiosMessage as field)
    if(field is simple)
      add element to output message, with value = field.value
    else
      add REST by example algorithm(element,field)
   mark field as used

  // if the element / attr. is not contained add the
  // element without value
  else
    add element without value to output message

if(there exists any non-marked field)
  throw error "incompatible"
else
  return output message
```

Listing 22: REST by example algorithm

G WSDL Description corresponding to Listing 11

```
<wsdl:types>
  <!-- namespace declarations omitted -->
  <xs:schema>
      <xs:complexType>
        <xs:sequence>

          <!-- simple string field 'name' -->
          <xs:element name="name" nillable="false"
            type="xs:string" />

          <!-- simple integer field 'age' -->
          <xs:element name="age" nillable="true"
             type="xs:int" />

          <!-- complex field 'address' -->
          <xs:element name="address" nillable="true">
             <xs:complexType>
               <xs:sequence>
                 <xs:element name="city" nillable="true"
                    type="xs:string" />
                 <xs:element name="house" nillable="true"
                    type="xs:int" />
                 <xs:element name="door" nillable="true"
                    type="xs:int" />
               </xs:sequence>
             </xs:complexType>
          </xs:element>

          <!-- string array field 'friends' -->
          <xs:element name="friends" nillable="true"
             type="xsd:string" maxOccurs="unbounded" />

        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
</wsdl:types>
```

Listing 23: WSDL description corresponding to Listing 11

H Input Stream Data Source used in Daios

```java
public class InputStreamDataSource implements OMDataSource {

  private InputStream is = null;

  // constructor
  public InputStreamDataSource(InputStream input) {
    this.is = input;
  }

  // serialize stream to output stream
  public void serialize(OutputStream output, OMOutputFormat format)
    throws XMLStreamException {

    XMLStreamWriter xmlStreamWriter =
      StAXUtils.createXMLStreamWriter(output);
    serialize(xmlStreamWriter);
    xmlStreamWriter.flush();

  }

  // serialize stream to writer
  public void serialize(Writer writer, OMOutputFormat format)
    throws XMLStreamException {

    serialize(StAXUtils.createXMLStreamWriter(writer));
  }

  // serialize stream to XML writer
  public void serialize(XMLStreamWriter xmlWriter)
    throws XMLStreamException {

    StreamingOMSerializer serializer = new StreamingOMSerializer();
    serializer.serialize(getReader(), xmlWriter);

  }

  // create a new reader from the data source input stream
  public XMLStreamReader getReader() throws XMLStreamException {

    XMLStreamReader reader = StAXUtils.createXMLStreamReader(is);
    return reader;

  }

}
```

Listing 24: Input Stream Data Source for AXIOM

I Complete Daios SOAP Example

```
String blz = ... // blz is a german bank identification number

// let's use the native backend
ServiceFrontendFactory factory = ServiceFrontendFactory.getFactory
    ("at.ac.tuwien.infosys.dsg.daiosPlugins."+
      nativeInvoker.NativeServiceInvokerFactory");

// preprocessing
// (This is a SOAP-based Web service that takes a german
//  BLZ and returns the bank details to this institute)
ServiceFrontend frontend = factory.createFrontend(
 new URL(
   "http://www.thomas-bayer.com/axis2/services/BLZService?wsdl"));

// construct message
DaiosInputMessage in = new DaiosInputMessage();
in.setString("blz", blz);

// do blocking invocation
DaiosOutputMessage out = frontend.requestResponse(in);

// convert WS result back into some convenient Java format
BankResult bank = new BankResult(blz);
bank.setBic(
  out.getComplex("details").getString("bic"));
bank.setName(
  out.getComplex("details").getString("bezeichnung"));
```

Listing 25: Complete example of Daios SOAP invocation

J Complete Daios REST Example

```
String myAPIKey = ... // you can get an API key from Flickr for free

// let's use the native backend
ServiceFrontendFactory factory = ServiceFrontendFactory.getFactory
    ("at.ac.tuwien.infosys.dsg.daiosPlugins."+
      nativeInvoker.NativeServiceInvokerFactory");

// preprocessing (no interface definitions means a RESTful
//                service without example request)
ServiceFrontend frontend = factory.createFrontend();

// setting the EPR is mandatory for this type of service
// (This is the Flickr REST interface)
frontend.setEndpointAddress(
  new URL("http://api.flickr.com/services/rest/"));

// construct message
DaiosInputMessage in = new DaiosInputMessage();
in.setString("method", "flickr.interestingness.getList");
in.setString("api_key", myAPIKey);
in.setInt("per_page", 5);

// do non-blocking invocation
PollObject po = frontend.pollObjectCall(in);

// ... now we do some other stuff while the invocation
//     is carried out in the background

// OK, let's look for the result
if(!po.responseReceived())
   // no result yet -
   // continue waiting / doing other stuff
else {

  DaiosOutputMessage out = po.getResult();
  DaiosMessage[] photos = out.getComplex("photos")
    .getComplexArray("photo");

  // now we could again convert the array of
  // photos into some nice Java format

}
```

Listing 26: Complete example of Daios REST invocation

K A Logging Interceptor for Daios

```
// create factory as usual
ServiceFrontendFactory factory = ...

// create frontend with an (anonymous) HTTP logger
ServiceFrontend frontend = factory.createFrontend(
  new URL("my.example.com/wsdl"),
  new DefaultInterceptor() {

    public void doHTTPInvocation(String endpoint, String body,
      IServiceFrontendImplementor invoker) {

      System.out.println("HTTP Request: ");
      System.out.println(body);

    }

    public void receiveHTTPResult(String endpoint, String body,
      IServiceFrontendImplementor invoker, String response) {

      System.out.println("HTTP Response: ");
      System.out.println(response);

    }

  });

// ... use frontend as usual
```

Listing 27: Logging SOAP payload using a Daios interceptor

L Performance Comparison Results

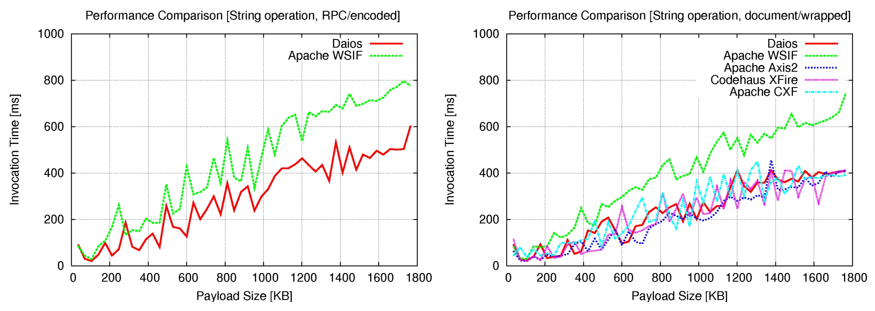


Figure 26: Comparison of simple invocations

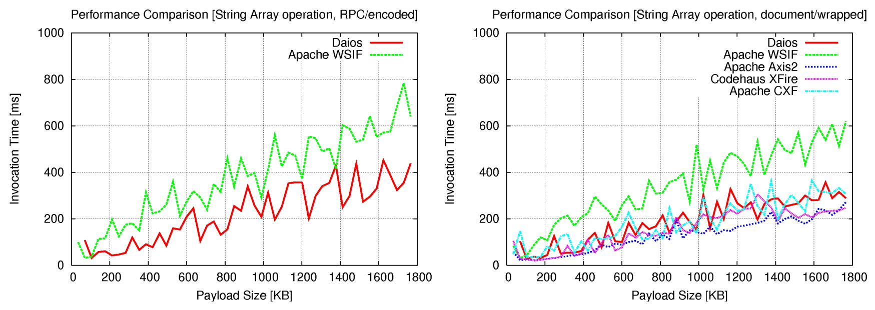


Figure 27: Comparison of array invocations

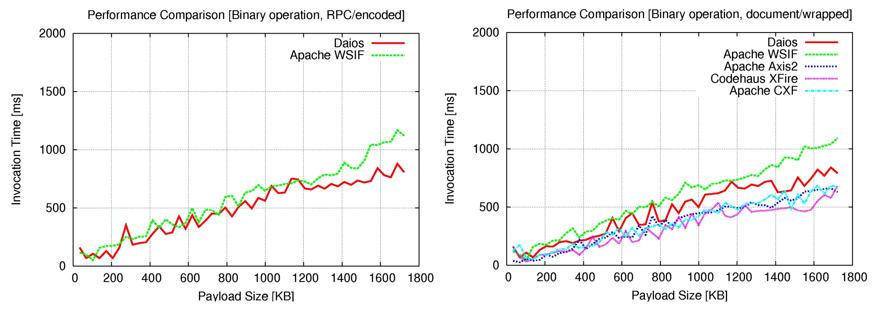


Figure 28: Comparison of binary invocations

References

[1] Nayef Abu-Ghazaleh, Michael J. Lewis, and Madhusudhan Govindaraju. Differential Serialization for Optimized SOAP Performance. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, 2004.

[2] Apache Foundation. Apache AXIOM. `http://ws.apache.org/commons/axiom/index.html`. Visited: 2007-07-28.

[3] Apache Foundation. Apache Axis. `http://ws.apache.org/axis/`. Visited: 2007-07-27.

[4] Apache Foundation. Apache Axis 2. `http://ws.apache.org/axis2/`. Visited: 2007-07-27.

[5] Apache Foundation. Apache CXF: An Open Source Service Framework. `http://incubator.apache.org/cxf/`. Visited: 2007-08-12.

[6] Apache Foundation. Web Services Invocation Framework. `http://ws.apache.org/wsif/`. Visited: 2007-07-28.

[7] Apache Foundation. XMLBeans. `http://xmlbeans.apache.org/`. Visited: 2007-09-04.

[8] Guruduth Banavar, Tushar Deepak Chandra, Robert E. Strom, and Daniel C. Sturman. A Case for Message-Oriented Middleware. In *Proceedings of the 13th International Symposium on Distributed Computing*, 1999.

[9] Tim Berners-Lee, Roy Fielding, and Larry Masinter. RFC 3986, Uniform Resource Identifier (URI): Generic Syntax. `http://rfc.net/rfc3986.html`, 2005. Visited: 2007-07-31.

[10] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. In *SOSP '83: Proceedings of the ninth ACM symposium on Operating Systems Principles*, 1983.

[11] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distributed and Abstract Types in Emerald. *IEEE Transanctions on Software Engineering*, 13(1), 1987.

[12] Paul Buhler, Christopher Starr William H. Schroder, and José M. Vidal. Preparing for Service-Oriented Computing: A Composite Design Pattern for Stubless Web Service Invocation. In *International Conference on Web Engineering*, 2004.

[13] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture, Volume 4 : A Pattern Language for Distributed Computing.* Wiley, 2007.

[14] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* Wiley, 1996.

[15] Russel Butek. Which style of WSDL should I use? `http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/`. Visited: 2007-08-02.

[16] Min Cai, Shahram Ghandeharizadeh, Rolfe R. Schmidt, and Saihong Song. A Comparison of Alternative Encoding Mechanisms for Web Services. In *DEXA '02: Proceedings of the 13th International Conference on Database and Expert Systems Applications*, 2002.

[17] The Castor Project. `http://www.castor.org/`. Visited: 2007-09-04.

[18] David Chappell. *Enterprise Service Bus.* O'Reilly, 2004.

[19] Christophe Demarey and Gael Harbonnier and Romain Rouvoy and Philippe Merle. Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms. *Studia Informatica Universalis Regular Issue*, 4(1), 2005.

[20] Codehaus. The Streaming API for XML (StAX). `http://stax.codehaus.org/`. Visited: 2007-07-29.

[21] Codehaus. XFire. `http://xfire.codehaus.org/`. Visited: 2007-08-12.

[22] William R. Cook and Janel Barfield. Web Services versus Distributed Objects: A Case Study of Performance and Interface Design. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*, 2006.

[23] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2), 2002.

[24] Francisco Curbera, Rania Khalaf, Nirmal Mukhi, Stefan Tai, and Sanjiva Weerawarana. The next Step in Web services. *Communications of the ACM*, 46(10), 2003.

[25] Matthew J. Duftler, Nirmal K. Mukhi, Aleksander Slominski, and Sanjiva Weerawarana. Web Services Invocation Framework (WSIF). In *Proceedings of the OOPSLA Workshop on Object-Oriented Web Services*, 2001.

[26] Schahram Dustdar and Martin Treiber. A View Based Analysis on Web Service Registries. *Distributed Parallel Databases*, 18(2), 2005.

[27] Robert Elfwing, Ulf Paulsson, and Lars Lundberg. Performance of SOAP in Web Service Environment Compared to CORBA. In *APSEC '02: Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, 2002.

[28] Wolfgang Emmerich. *Engineering Distributed Objects*. Wiley, 2000.

[29] Thomas Erl. *Service-Oriented Architecture. Concepts, Technology, and Design*. Prentice Hall, 2005.

[30] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2), 2003.

[31] Eviware. soapUI. `http://www.soapui.org/`, 2007. Visited: 2007-08-13.

[32] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Peter Leach, and Tim Berners-Lee. RFC 2616, Hypertext Transfer Protocol – HTTP/1.1. `http://www.rfc.net/rfc2616.html`, 1999. Visited: 2007-07-31.

[33] Roy T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, CA, 2000.

[34] Roy T. Fielding and Richard N. Taylor. Principled Design of the modern Web Architecture. *ACM Transactions on Internet Technology*, 2(2), 2002.

[35] Flickr. `http://www.flickr.com/`, 2007. Visited: 2007-08-14.

[36] Flickr Services - API documentation. `http://www.flickr.com/services/api/`, 2007. Visited: 2007-08-14.

[37] Organization for the Advancement of Structured Information Standards (OASIS). RELAX NG specification. `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`, 2001. Visited: 2007-07-27.

[38] Organization for the Advancement of Structured Information Standards (OASIS). OASIS/ebXML Registry Services Specification v2.0. `http://www.oasis-open.org/committees/regrep/documents/2.0/specs/ebrs.pdf`, 2002. Visited: 2007-07-31.

[39] Organization for the Advancement of Structured Information Standards (OASIS). Reference Model for Service Oriented Architecture 1.0. `http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf`, 2006. Visited: 2007-07-31.

[40] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Business Process Execution Language Version 2.0, Draft. `http://www.oasis-open.org/committees/download.php/18714/wsbpel-specification-draft-May17.htm`, 2006. Visited: 2007-07-27.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[42] David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1), 1985.

[43] Karl Gottschalk, Stephen Graham, Heather Kreger, and James Snell. Introduction to Web services Architecture. *IBM Systems Journal*, 41(2), 2002.

[44] JSR-101 Expert Group. Java API for XML-Based RPC, Version 1.1. `http://java.sun.com/xml/downloads/jaxrpc.html#jaxrpcspec10`, 2003. Visited: 2007-08-08.

[45] THALES Group. Service Registries Study. `http://www.chatelp.org/work/LUCAS_registry_study.pdf`, 2006. Visited: 2007-07-31.

[46] Marc J. Hadley. Web Application Description Language (WADL). `https://wadl.dev.java.net/wadl20061109.pdf`, 2006. Visited: 2007-07-27.

[47] Gregor Hohpe and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley, 2003.

[48] Michael N. Huhns and Munindar P. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1), 2005.

[49] Steve Jones. Toward an Acceptable Definition of Service. *IEEE Software*, 22(3), 2005.

[50] Doug Kohlert and Arun Gupta. Java API for XML-Based Web Services, Version 2. `http://jcp.org/aboutJava/communityprocess/mrel/jsr224/index2.html`, 2007. Visited: 2007-08-08.

[51] Takashi Koshida and Shunsuke Uemura. Automated Dynamic Invocation System for Web Service with a User-defined Data Type. In *Proceedings of the 2nd European Workshop on Object Orientation and Web Services*, 2004.

[52] Gabriel M. Kuper and Jérôme Siméon. Subsumption for XML types. In *ICDT '01: Proceedings of the 8th International Conference on Database Theory*, 2001.

[53] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing (PODC)*, 1986.

[54] David S. Linthicum. *Enterprise Application Integration.* Addison-Wesley, 2000.

[55] Piyush Maheshwari, Trung Nguyen Kien, and Abdelkarim Erradi. QoS-Based Message-Oriented Middleware for Web Services. In *WISE Workshops*, 2004.

[56] Piyush Maheshwari, Hua Tang, and Roger Liang. Enhancing Web Services with Message-Oriented Middleware. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, 2004.

[57] Shinichi Nagano, Tetsuo Hasegawa, Akihiko Ohsuga, and Shinichi Honiden. Dynamic Invocation Model of Web Services Using Subsumption Relations. In *ICWS '04: Proceedings of the IEEE International Conference on Web Services (ICWS'04)*, 2004.

[58] ObjectWeb. Celtix: The Open Source Java Enterprise Service Bus. `http://celtix.objectweb.org/`. Visited: 2007-08-12.

[59] Michael P. Papazoglou and Willem-Jan van den Heuvel. Service-Oriented Design and Development Methodology. *International Journal of Web Engineering and Technology (IJWET)*, 2(4), 2006.

[60] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented Architectures: Approaches, Technologies and Research Issues. *The VLDB Journal*, 16(3), 2007.

[61] Mike.P. Papazoglou. Service -Oriented Computing: Concepts, Characteristics and Directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE)*, 2003.

[62] Mike.P. Papazoglou and Dimitrios Georgakopoulos. Service-oriented Computing. *Communications of the ACM*, 46(10), 2003.

[63] Savas Parastatidis, Simon Woodman, Jim Webber, Dean Kuo, and Paul Greenfield. Asynchronous Messaging between Web Services Using SSDL. *IEEE Internet Computing*, 10(1), 2006.

[64] Graham D. Parrington. Reliable Distributed Programming in C++: The Arjuna Approach. In *C++ Conference*, 1990.

[65] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, 2007.

[66] Dirk Riehle. Composite Design Patterns. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1997.

[67] Jothy Rosenberg and David Remy. *Securing Web Services with WS-Security - Demystifying WS-Security, WS-Policy, SAML, XML Signature, and XML Encryption*. Pearson, 2004.

[68] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 2005.

[69] Douglas C. Schmidt, Hans Rohnert, Michael Stal, and Dieter Schultz. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

[70] M.-T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen. The Enterprise Service Bus: making Service-Oriented Architecture real. *IBM Systems Journal*, 44(4), 2005.

[71] Simple Asynchronous Invocation Framework for Web Services. `http://saiws.sourceforge.net/`, 2003. Visited: 2007-08-01.

[72] Aleksander Slominski. Design of a Pull and Push Parser System for Streaming XML. `http://www.extreme.indiana.edu/xgws/papers/xml_push_pull.pdf`, 2002. Visited: 2007-07-29.

[73] James Snell. Resource-oriented vs. activity-oriented Web Services. `http://www-128.ibm.com/developerworks/webservices/library/ws-restvsoap/`, 2004. Visited: 2007-07-27.

[74] SOAP::Lite for Perl. `http://www.soaplite.com/`. Visited: 2007-07-29.

[75] Toyotaro Suzumura, Toshiro Takase, and Michiaki Tatsubori. Optimizing Web Services Performance by Differential Deserialization. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, 2005.

[76] Stefan Tai, Thomas A. Mikalsen, and Isabelle Rouvellou. Using message-oriented middleware for reliable web services messaging. In *Second International Workshop on Web Services, E-Business, and the Semantic Web*, 2003.

[77] Andrew Tanenbaum and Marten van Steen. *Distributed Systems: Principles and Paradigms, 2/E*. Prentice Hall, 2006.

[78] Alfred Lord Tennyson. *Tennyson: Including Lotos Eaters, Ulysses and Others*. Kessinger, 2004.

[79] John R. R. Tolkien. *The Lord of the Rings - 50th Anniversary Single Volume Edition*. Harpercollins, 2005.

[80] Aphrodite Tsalgatidou and Thomi Pilioura. An Overview of Standards and Related Technology in Web Services. *Distributed and Parallel Databases*, 12(2-3), 2002.

[81] UDDI.org. UDDI Technical White Paper. `http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf`, 2000. Visited: 2007-07-31.

[82] Markus Voelter, Michael Kircher, and Uwe Zdun. *Remoting Patterns - Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*. Wiley, 2005.

[83] Werner Vogels. Web Services Are Not Distributed Objects. *IEEE Internet Computing*, 7(6), 2003.

[84] World Wide Web Consortium (W3C). WSDL, Web Service Description Language. `http://www.w3.org/TR/wsdl`, 2002. Visited: 2007-07-31.

[85] World Wide Web Consortium (W3C). SOAP Version 1.2 Part0: Primer. `http://www.w3.org/TR/soap12-part0/`, 2003. Visited: 2007-07-31.

[86] World Wide Web Consortium (W3C). Web Services Addressing (WS-Addressing). `http://www.w3.org/Submission/ws-addressing/`, 2004. Visited: 2007-07-31.

[87] World Wide Web Consortium (W3C). Web Services Choreography Description Language Version 1.0, W3C Working Draft. `http://www.w3.org/TR/ws-cdl-10`, 2004. Visited: 2007-07-31.

[88] World Wide Web Consortium (W3C). SOAP Message Transmission Optimization Mechanism. `http://www.w3.org/TR/soap12-mtom/`, 2005. Visited: 2007-07-28.

[89] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0 Part 0: Primer - W3C Candidate Recommendation 27 March 2006. `http://www.w3.org/TR/2006/CR-wsdl20-primer-20060327/`, 2006. Visited: 2007-07-31.

[90] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A Note on Distributed Computing. Technical report, Sun Microsystems Labs, 1994.

[91] Web service interoperability organization (WS-I). Basic Profile Version 1.2. `http://www.ws-i.org/Profiles/BasicProfile-1.2.html`. Visited: 2007-08-02.

[92] Peter Wegner. Concepts and Paradigms of object-oriented Programming. *SIGPLAN OOPS Messenger*, 1(1), 1990.

[93] Mark Weiser. Ubiquitous Computing. *IEEE Computer*, 26(10), 1993.

[94] Uwe Zdun, Markus Voelter, and Michael Kircher. Design and Implementation of an Asynchronous Invocation Framework for Web Services. In *ICWS-Europe*, 2003.

[95] Uwe Zdun, Markus Voelter, and Michael Kircher. Pattern-Based Design of an Asynchronous Invocation Framework for Web Services. *International Journal of Web Service Research*, 1(3), 2004.